Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

On-Board Software Reference Implementation for Space Exploration Systems

Master's Thesis

by

Jens Kolbenschlag

June 15, 2025

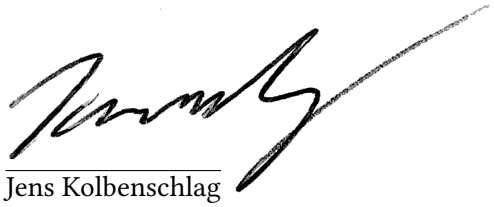
University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Klaus Schneider, Prof. Dr.
Artur Scholz, PhD, M.Sc., Dipl.-Ing.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „On-Board Software Reference Implementation for Space Exploration Systems“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 15.6.2025



Jens Kolbenschlag

Abstract

In this thesis, we present a reference implementation of spacecraft onboard software that provides interfaces for onboard equipment, application software, and mission control systems. This represents the first open-source implementation demonstrating the integration of multiple European standards for onboard architecture and communication protocols. The system is extended by novel approaches, such as the use of a redundant CAN bus as the onboard communication bus, supporting fault-tolerant and scalable configurations. The implementation is Python-based and follows a modular design philosophy, enabling simplified development of both hardware and software components, and serving as a reference for reliable embedded software development in the space domain. We demonstrate the successful integration of the standards into a functional software stack, which was validated through an integrated system test using representative onboard components and communication chains. This work contributes a reproducible and extensible foundation for CubeSat missions and demonstrates the integration of space agency standards in small satellite developments.

Usage of Generative AI

Parts of this thesis were written with the assistance of the language model ChatGPT by OpenAI. The AI was used as a tool to support the drafting process through language generation, rephrasing, or stylistic suggestions. All AI-assisted content has been critically reviewed, edited, and verified by the author to ensure accuracy, academic integrity, and compliance with the standards of scholarly work.

Contents

1. Introduction

The landscape of space exploration has been revolutionized by the emergence of CubeSats – standardized, miniature satellites in cubic units of 10x10x10 cm [Cub22]. These compact and cost-effective platforms have significantly broadened access to space, moving beyond the traditional domain of large governmental space agencies. Today, CubeSats are developed by a diverse array of organizations, including universities and educational institutions worldwide, who leverage them for hands-on student learning and cutting-edge research [Kul24]. Beyond academia, private companies and commercial enterprises are increasingly investing in CubeSat development for various applications, from Earth observation and remote sensing to communications and in-orbit technology demonstrations. Furthermore, government organizations and research laboratories also utilize CubeSats for scientific investigations, technology validation, and as a low-cost means to deploy experimental payloads [Kul24].

1.1. General Problem Setting

In recent years, CubeSat development has risen significantly in popularity with several hundred deployments per year [Kul24]. However, the development of satellite systems and software remains a complex and costly task. In this work, we propose a reference implementation of a spacecraft Onboard Software (OBSW) specifically tailored towards the specific needs of a CubeSat mission. This software stack was developed with the intention to be used as both a reference implementation for demonstration purposes, but also to form a functioning system for testing of the newly developed CubeSat hardware platform by LibreCube (see Section 1.2).

Onboard Software Introduction

The term “Onboard Software” formally encompasses all software executing within a spacecraft. This broad definition includes equipment firmware, network drivers, operating systems, and higher-level applications, which are often distributed across various hardware units onboard the spacecraft [Eic12]. In this work, our primary focus is on the Data Handling System (DHS) portion of the OBSW, specifically the software that runs on the spacecraft’s Onboard Computer (OBC).

This system plays a crucial role in the spacecraft’s operation, responsible for a multitude of tasks. These include the collection of scientific data and parameters from onboard equipment, the execution of automations based on the spacecraft’s current

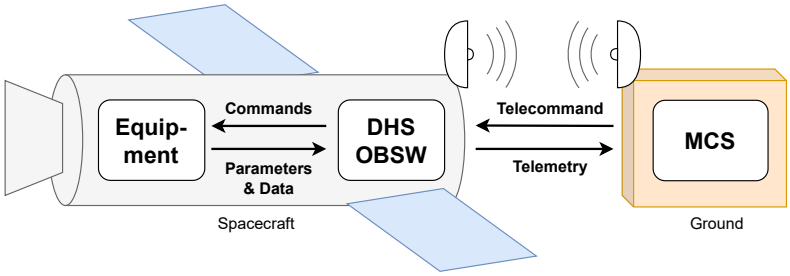


Figure 1.1.: *Onboard Software Overview*

state, and the direct execution of commands received from ground control. Furthermore, the DHS manages the storage of all collected data, reports the spacecraft's health and status to ground stations, and transmits scientific data back to Earth. Finally, it is vital for the monitoring and recovery of the spacecraft's overall health [Eic12].

The DHS interfaces the onboard-bus to communicate with onboard equipment, such as scientific instruments, the Attitude and Orbit Control System (AOCS), the Power Control and Distribution Unit (PCDU) and other hardware. For communication with the ground-based Mission Control System (MCS), it is also connected to the antenna communication system, usually through a separate bus with higher bandwidth. From ground it receives commands in the form of Telecommand (TC) packets, and returns data in Telemetry (TM) packets. These packets are encoded by a chain of different protocols responsible for segmentation, routing, data link control, error control and finally physical modulation into radio waves. The OBSW must be able to both decode this protocol chain for telecommand, and to encode telemetry accordingly [Eic12].

Space systems usually require a common set of features, such as the generation and modification of housekeeping reports, or executing onboard automation scripts. For this reason the service-based software architecture became popular in the industry, where certain feature-sets are integrated into services, which are then commandable from ground with specifically defined TC packet structures [Eic12], [Eur16]. Our proposed implementation follows this service-based approach for the DHS and even expands it to be used over the system-bus for communication with equipment.

Our Proposal

Today, several open-source frameworks for onboard software development in small satellites are available. Most of these frameworks decouple the software from the underlying hardware by means of a Hardware Abstraction Layer (HAL), thereby enabling portability across different hardware architectures. Typically, such frameworks employ a custom software architecture, are centered around a specific programming language – most commonly C++ – and define strict conventions for how mission-specific software components must be implemented. These design decisions are often highly opinionated and can vary significantly between frameworks.

We intended to pursue a different approach. Rather than developing and publishing yet another framework for onboard software, our objective is to provide a reference implementation based on standardized architectures and interfaces between spacecraft onboard components. Our aim is to encourage developers to adopt an interface-driven design for their CubeSat missions. As demonstrated by European Space Agency (ESA) and other space agencies, the adoption of international space standard protocols enables cross-support (e.g., ground stations tracking different missions) and reusability (e.g., employing a single mission control system for multiple missions). Furthermore, by adopting established architectures developed by space agencies, CubeSat developers can benefit from the reliability and robustness that are inherent in these well-established standards [SM23].

Namely, the standards we intended to follow and implement are:

- **Space Avionics Open Interface Architecture (SAVOIR)**: Reference architecture for satellite systems, published by ESA [Eur21].
- **Packet Utilization Standard (PUS)**: Standardized onboard services and corresponding application layer protocol based on space packets, published by the European Cooperation for Space Standardization (ECSS) [Eur16].
- **CCSDS File Delivery Protocol (CFDP)**: Application layer protocol for file transfer, published by the Consultative Committee for Space Data Systems (CCSDS) [Con20a].
- **Space Packet**: Network layer protocol used by all major space agencies, published by the CCSDS [Con20b].
- **CCSDS Frames**: Data link layer protocol for segmentation, sequence and error control, published by the CCSDS [Con21b], [Con21a].

We will discuss these standards in detail in Chapter 2.

Related Work

Two widely used open-source onboard software stacks are the F Prime (F´) framework [Jet] and the Core Flight System (cFS) [Nat], both of which have been developed and released by NASA.

The F´ framework adopts a component-oriented, modular architecture, allowing for high configurability and ease of reuse. It comes with a comprehensive set of pre-developed components that cover many of the basic functionalities required for spacecraft onboard software, thereby reducing development time and effort.

The cFS, on the other hand, is based on a software bus architecture that follows a publish/subscribe pattern. It provides a large library of reusable applications and protocols that are readily available for integration into a mission-specific software stack. This architecture facilitates loose coupling between components and improves scalability and maintainability.

Both frameworks are implemented in C or C++ and offer an abstraction layer that

decouples the application logic from the underlying operating system.

Despite their maturity and broad adoption, these frameworks rely on the use of custom application-layer protocols that are specific to their ecosystems. Notably, they do not natively support European standards, such as those defined by the ECSS and ESA, for standardized monitoring and control services. This lack of compliance can present integration challenges for missions aiming to adhere to ESA conventions or to interoperate with ground systems and components based on ECSS standards.

1.2. New Contributions

As discussed in Section 1.1, our main goal is to follow and implement public international, and especially European standards. In this section, we will briefly introduce aspects of our OBSW implementation that are either complete novelties, or standards that were defined but not (yet) used in our proposed way by the industry.

LibreCube Hardware Platform

LibreCube is a german registered association with the goal to develop open-source hardware and software for CubeSats, focused on following existing standards to make them more accessible to a broad audience [Lib]. As of the time of writing, LibreCube has developed a modular Printed Circuit Board (PCB) design to be used in CubeSats, which is currently being tested in a fully integrated rover called *LibreCubeRover* (see Figure 1.2). A rudimentary remote control software was developed for the rover to demonstrate basic hardware functionality, but it offers no support for space protocol compatibility, robustness or reusability. The OBSW developed in this work is intended to be a fully functional onboard software, in order to control this rover through common space protocols and to demonstrate the successful integration of all systems.

The components of the rover are described by LibreCube as follows [Lib]:

> The basic system components of the rover are:
>
> - Structure: It provides housing for the electronic board stack and fixtures for the wheels.
> - Power Module: It supplies the rover with electrical power from batteries. Can be re-charged.
> - Communications Module: For wireless data exchange with the rover.
> - Processing Module: The onboard computer manages the rover and is the central interface.
> - Wheel Drive Module: Controls the movement of the wheels.
>
> The additional system components are:

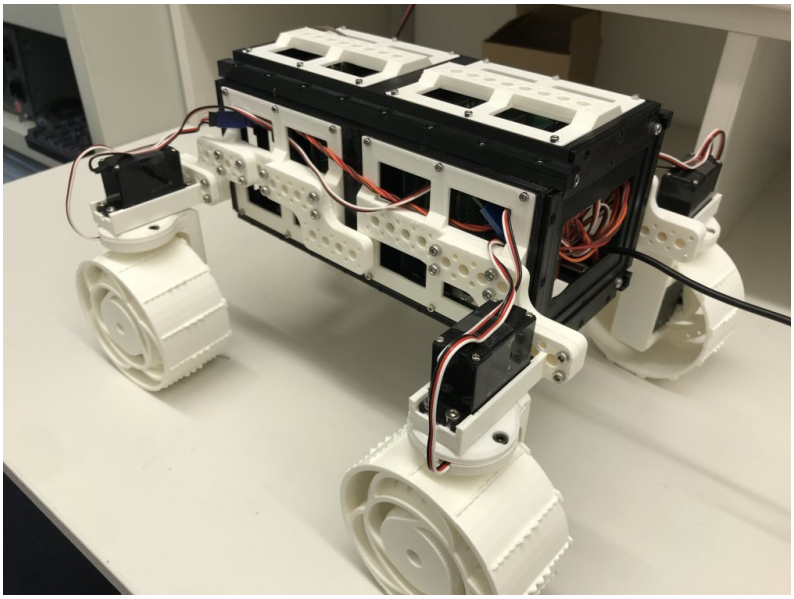


Figure 1.2.: *LibreCubeRover, Source: [Lib]*

- Camera Module: To capture images, possible stream video.
- Solar Panels: Provide charging power for the rover.
- Battery Expansion Module: Extends the battery capacity of the rover.
- Navigation Module: Provides position information and heading.

The board dimensions follow the *CubeSat Design Specification* [Cub22], which defines a standard form factor for CubeSat modules. This ensures compatibility with a wide range of CubeSat deployers and facilitates integration with other standardized components. Each module is designed to fit within the 100 mm × 100 mm footprint and specified height constraints, allowing for modular stacking and easy expansion of system capabilities. For the electrical integration of the PCBs into a modular stack, the design follows the *PC/104 Specification* [PC108]. It defines a 104 Pin board-to-board connector, as well as the mechanical mounting of multiple PCBs into a robust stack.

LibreCube's board specification allocates a defined set of pins of the 104 pin board-to-board connector for standard functionality that is shared across the different modules, such as communication buses and power lines. This allows for straightforward interchangeability, reusability and expandability of a system, even without the need to modify a given module design. The connector pin allocation intentionally reserves parts of the connector to be defined by the user for mission specific signals.

The design uses hot redundant[1] UART connections for TM and TC transmission between the antenna module and the OBC. For platform monitoring and commanding, it uses a novel warm redundant[1] CAN bus system called *SpaceCAN* [Sch+19], which we will introduce in Section 1.2 and discuss in detail in Section 2.3.

For hardware design, LibreCube prioritizes the use of standard components and off-the-shelf hardware to lower the hurdle for hardware designers to get into CubeSat development.

SpaceCAN as Onboard Bus

CubeSats historically rely on simple, non-redundant onboard communication protocols such as I^2C, SPI or UART [BLG17]. This however poses a problem, as these protocols do not offer error detection, error correction or failure recovery mechanisms out of the box. While radiation levels on low earth orbits of typical CubeSat missions are still very low compared to deep space missions, the possibility of Single Event Upsets (SEUs) as a result of high energy particles or other problems induced by electromagnetic radiation remains a considerable threat [Lik+10]. In the event of a failure of an onboard system, the spacecraft must either be capable of autonomously switching to a redundant system or retain basic Monitoring and Control (M&C) functionality to enable ground controllers to reconfigure the spacecraft. The onboard communication bus is crucial to maintaining the spacecraft's functional integrity and recovery capability. Consequently, a failure in the onboard bus often results in the failure of the CubeSat mission in practice [BLG17].

In professional satellite missions with high budgets, such as missions from ESA or NASA, dedicated robust bus systems are used such as *MIL-STD-1553* or *SpaceWire* [Eura], [Sch+19]. However, these bus systems require specialized hardware and software that is not readily available on consumer markets so that they are deemed not feasible for most CubeSat missions.

The CAN bus offers several advantages for use as an onboard communication bus, including built-in error detection and resilience to electromagnetic interference, achieved through the use of an isolated differential signal transmitted via a twisted pair connection. Its robustness has been demonstrated by its widespread adoption in the automotive industry, where it has been employed for many years to interconnect various vehicle subsystems. Nevertheless, the CAN bus remains susceptible to electrical failures, such as issues with physical connections or malfunction of the transceivers responsible for converting the differential CAN signal into a CMOS-level signal suitable for interfacing with microcontrollers. To qualify the CAN bus for space applications, the ECSS has defined a redundant extension of the standard CAN protocol in the specification *ECSS-E-ST-50-15C* [Eur15], commonly referred to as *ECSS-CAN*. This standard also introduces a service-oriented meta-protocol to enhance communication reliability and structure.

However, LibreCube deemed the ECSS-CAN to be too complex to be used and implemented in small spacecraft, which is why a simpler, but functionally similar protocol *SpaceCAN* was developed in 2019 [Sch+19]. It is also based on two redundant

[1] We will use the redundancy terms of the SAVOIR documents [Eur21], which are defined as follows:
- Cold redundancy: One active part, redundant part is not operating.
- Warm redundancy: One active part, redundant part is operating and ready to take over.
- Hot redundancy: All parts are actively operating in parallel.

CAN-bus systems, that can automatically detect a failure and recover to the other bus, and it also supports the use of service-based communication similar to PUS. We will discuss the protocol in detail in Section 2.3.

Open-source SAVOIR Implementation

The Space Avionics Open Interface Architecture (SAVOIR) initiative, led by ESA in collaboration with European space industry partners, aims to standardize and harmonize avionics systems for spacecraft. It addresses the increasing complexity and cost associated with the development of onboard systems by promoting a set of common building blocks, interfaces, and design principles for spacecraft avionics [Eur21].

SAVOIR provides a reference architecture for developing avionics subsystems, including onboard computers, data handling units, software architecture and communication interfaces. By defining reusable functional components and standardizing key interfaces, the initiative enhances modularity, interoperability, and maintainability across missions. This approach not only reduces development time and risk but also supports long-term sustainability and reusability of avionics designs.

The SAVOIR recommendations have become increasingly influential in European space projects, especially for institutional and commercial satellite platforms. The architecture covers both the functional decomposition of avionics systems and guidance for implementing protocols, such as the Packet Utilization Standard (PUS). The documents are available free of charge on the *European Space Software Repository* [Eurb] for institutions in the ESA member states, but distribution to other states requires the approval of the Industrial Policy Committee (IPC).

In the context of this thesis, SAVOIR serves as a foundational framework to guide the organization, interface design, communication principles and architecture of our onboard system and software. In particular, this work will focus on two documents:

- **SAVOIR Functional Reference Architecture (FRA)**:
  A reference architecture describing the functional units commonly required onboard a reliable space system.
- **SAVOIR Onboard Software Reference Architecture (OSRA)**:
  A reference architecture for onboard software design.

To the best of our knowledge, this thesis presents the first open-source implementation based on the SAVOIR architecture.

Onboard Procedures with PLUTO

Onboard Control Procedures (OBCPs) are automation scripts on board of a spacecraft system, to be executed in space. The onboard automation stands in contrast to the still more usual ground-based automation, which autonomously sends telecommand based on automation scripts from ground. But for most missions, there is

no continuous contact to ground, so there is the need of providing automation capability for the spacecraft system on its own. One option to make the spacecraft execute commands autonomously without ground contact, is to upload commands to the spacecraft in advance, together with the time at which thy should be executed. While this approach may be sufficient in some cases, more complex tasks pose the need for an automation system that supports control flow. This need is fulfilled by the onboard OBCP handling system.

OBCPs are scripts that can be executed onboard of a spacecraft, with access to the onboard system parameters, events and commands. They can be uploaded and modified from ground without the need of patching the whole OBSW during flight, and are intended to be easily understandable by the spacecraft operation team and can be patched in case some functionality has to be extended [Eic12]. Usually they are written in a domain-specific language that incorporates common monitoring and control processes, such as confirming correct system status before proceeding, or including timeouts for certain operations. There currently exists no suggested standard programming language for OBCPs, but only a standard that defines the capabilities of the OBCP handling system [Eur10].

On the other hand, the ECSS does define the scripting language PLUTO, which stands for "Procedure Language for Users in Test and Operations" [Eur08]. As the name and documentation implies, this language is mainly intended to be used in testing and ground-based mission control operations, where it gained popularity in recent years. However, the required functionality and spacecraft interaction of onboard procedures is very similar those of ground-based automation scripts. For this reason, and in an effort to simplify the development of automation scripts in general, we decided to experimentally implement an onboard OBCP handler that works with PLUTO scripts in our system. This approach would make it easier for the spacecraft operations team to develop and maintain OBCPs, as they are already familiar with the PLUTO language. To the best of our knowledge, PLUTO has so far not been used as an OBCP language in a spacecraft system.

File Transfer Coordination

One other, less significant novelty we implemented has to do with the latest extension, version C of the PUS standard [Eur16]. This extension defined system requirements more precisely, and also added new service definitions, such as a new service dedicated to file operations, Service 23 – *File Management*. Service 23 provides functionality to modify onboard file and folder entries, but also defines commands for copying and moving files. While the specifications hints at the usage of this service for initiating file transfers between ground and spacecraft, it does not provide further details about the actual implementation of this feature using only the defined commands. To the best of our knowledge, our work provides the first implementation of file transfer coordination using only the standard commands defined by PUS.

1.3. Outline of the Thesis

In Chapter 1 – *Introduction*, we introduced the overall goals and concepts that we intended to implement in the work, and put them into context within the aerospace industry.

In Chapter 2 – *Preliminaries*, we discuss detailed preliminary definitions of system requirements, architecture decisions, interfaces and communication protocols used in our implementation. This includes externally defined standards and software libraries we used, and provides reasoning for the design decisions of our implementation.

In Chapter 3 – *Main Contributions*, we present the core contributions of this work, namely the development of a new onboard software stack with all its submodules. Additionally, we present the development of an optional hardware module which integrates advanced reliability mechanisms of agency satellites into our CubeSat system.

In Chapter 4 – *Experimental Evaluation*, we discuss the performance of our implemented system based on a simple fully integrated system test, and compatibility to commonly used mission control software.

In Chapter 5 – *Conclusions*, we summarize our results and the limitations of our implemented system, and based on this evaluation we present opportunities of future work based on this thesis.

2. Preliminaries

2.1. Concerns and Requirements

In this section, we will define the architecture concerns and the derived architecture requirements, which guide our work towards a functional implementation. This process follows industry standard software architecture design approaches, such as described by the book *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* [RW05]. Although, as this thesis does not aim to provide a formal architecture documentation, we will follow this method and their definitions of the terms "concern" and "requirement" only in a generalized and informal manner.

We will reference these concerns and requirements throughout the following chapters to justify our design decisions, and to ensure traceability between concerns, requirements, design decisions and implementation.

Concerns

The concerns are derived from the ideas presented in Section 1.1 and express objectives and intends for the solution that our implementation shall provide. These concerns do not dictate decisions and are typically not quantifiable, but helped to steer our work towards the desired direction, as they did not change throughout the project's duration.

CON 1 – Onboard Software Stack
The software shall provide common functionality required on a typical CubeSat, such as M&C from ground, onboard data handling, onboard equipment interaction, onboard automation, and it should provide an abstraction layer for onboard applications.

CON 2 – Reference Implementation
The software shall pose as a reference implementation to guide developers on how to implement open space standards into an OBSW.

CON 3 – Testing Platform
The software should facilitate prototype development by offering a functional onboard software which can be easily adapted to different system configurations.

CON 4 – Compatibility
The system must be compatible with the available hardware platform by LibreCube.

CON 5 – Modifyability
The system shall allow typical tailoring towards mission specific applications.

CON 6 – Understandability
The implementation must be easy to understand.

CON 7 – Architecture Protability
It shall be possible to port the overall software architecture to different hardware and languages.

CON 8 – Use of Standards
The system shall adhere to available publicly defined standards and protocols and shall be compatible to systems that support the respective standards.

CON 9 – Restricted Development Time
The system must be designed, implemented, and evaluated within the six-month duration of the Master's thesis project.

Requirements

Based on the concerns, we derive the key requirements that give concrete constraints and provide reasoning towards making design decisions.

REQ 1 – Open-Source
The software shall be published in an open-source repository. *Derived from CON 2.*

REQ 2 – Reproducibilty
The software shall include example code and instructions to reproduce a simple functional system. *Derived from CON 2.*

REQ 3 – Hardware Interfaces
The software shall run on an embedded PC that can communicate via two CAN ports and two UART ports. *Derived from CON 4.*

REQ 4 – Modularity
Functional parts of the software shall define minimal interfaces towards the rest of the system, so that it allows exchange and reuse. *Derived from CON 3 and CON 5.*

REQ 5 – Customizeability
The software shall provide simple means to incorporate mission specific customization, such as onboard parameters, events, commands, equipment interaction. *Derived from CON 5.*

REQ 6 – Extendability
The software shall provide simple means to extend the software by mission specific applications functionality, such as equipment handlers, custom M&C services and onboard applications. *Derived from CON 5.*

REQ 7 – Use of the Python Language

The software shall be written in Python, as it is considered easy to understand, and offers a large selection of open-source libraries and relevant protocol implementations by LibreCube. *Derived from CON 6 and CON 9.*

REQ 8 – Use of Universal Features
The software architecture shall only rely on hardware and software features, that are supported by typical CubeSat OBC hardware platforms, specifically embedded PCs running Linux and microcontrollers running FreeRTOS. *Derived from CON 7.*

REQ 9 – Convention Compliance
The implementation shall conform to established industry best practices and design patterns. *Derived from CON 6 and CON 7.*

REQ 10 – SAVOIR-based Architecture
The architecture design and system implementation shall be closely oriented on the ESA Space Avionics Open Interface Architecture (SAVOIR). *Derived from CON 8.*

REQ 11 – Monitoring & Control with PUS
The software stack shall provide monitoring and control services and TC decoding / TM encoding according to the Packet Utilization Standard (PUS). *Derived from CON 8.*

REQ 12 – File Transmission with CFDP
The software stack shall implement the CFDP protocol for file transmission between ground and space system. *Derived from CON 8.*

REQ 13 – Onboard Bus with SpaceCAN
The software stack shall incorporate the SpaceCAN protocol for onboard bus communication with equipment. *Derived from CON 8.*

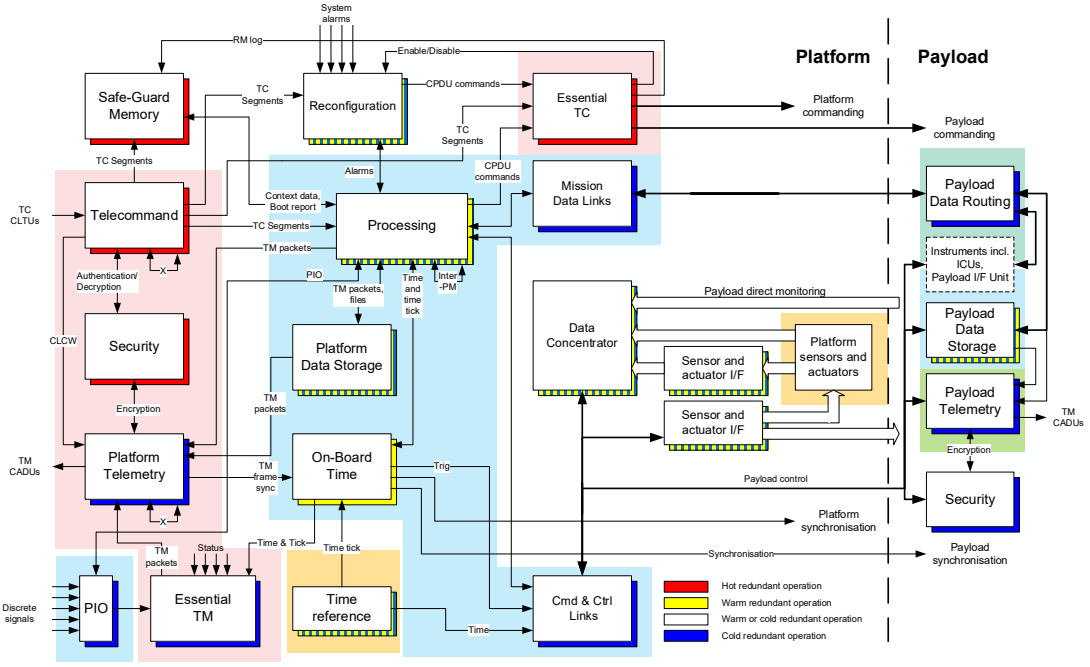
REQ 14 – Onboard Control Procedures with PLUTO
The software stack shall incorporate onboard procedure handling for scripts written in the PLUTO language. *Derived from CON 8.*

2.2. Avionics System Architecture Overview

Although the main focus of this thesis lies on the development of the onboard software, to get a better understanding of the required interfaces, we also looked into the architecture of the whole spacecraft avionics system. In particular, we analyzed the existing LibreCube hardware architecture in regards of interaction with the onboard software on the main computer, which we will call *Processing Unit*.

As a reference, we use the SAVOIR Functional Reference Architecture (FRA) (also known as Avionics System Reference Architecture (ASRA)), which was defined by ESA and reflects the functional structure of onboard avionics systems of modern ESA satellites [Eur21]. However, CubeSats usually have different requirements compared to “bigger” satellites built by the major space agencies, and as such the FRA does not fully apply to CubeSats. They have limited financial budgets, space, power,

operate in a less demanding environment, and handle lower volumes of scientific data, and thus don't require some advanced features of agency satellite avionics. Most CubeSats only stay in low earth orbit, where the risk of SEUs due to radiation is comparatively low, so the hardware requirements are less strict and allow for usage of off-the-shelf hardware [SM23]. In 2023, Szewczyk et al. [SM23] from ESA published a concept on how to apply the SAVOIR FRA to CubeSat applications, including a proposed hardware allocation for the required functional blocks. The functional overview, together with this hardware allocation proposed by ESA for CubeSats is shown in Figure 2.1.



CubeSat	SAVOIR
OBC/Processing Module	○ Processing ○ Platform Data Storage ○ Payload Data Storage ○ On-board Time ○ Mission Data Links ○ Cmd & Ctrl Links ○ (Optional) Payload processing ○ Parallel I/O
COMMS	○ Telecommand ○ Platform Telemetry ○ Essential TC/Essential TM (by direct CAN connection to modules) ○ Security (optional, for TC authentication/decryption)
Power Conditioning and Distribution Module (PCDM)	○ Power control & distribution ○ Reconfiguration (limited. Can power cycle modules via CAN bus) ○ Parallel I/O (if MCU present)
AOCS Incl. GPS module	○ Time reference ○ Position and velocity sensor ○ AOCS sensors and actuators
Payload TM (optional)	Payload Telemetry
-	Payload Security
(Distributed function)	Data Concentrator (RTU)
-	Sensor & actuator I/F
(Partial, in COMMS and PCDM)	Reconfiguration
-	Safe Guard Memory

Figure 2.1.: *SAVOIR Functional Reference Architecture and Mapping to CubeSat Functional Blocks. Source: [SM23]*

The FRA is divided into two sides: The *Platform* side and the *Payload* side.

The *Platform* side provides critical functionality for the spacecraft system in regards of communication (both with ground and onboard), data handling, automation, and Failure Detection, Isolation and Recovery (FDIR). Functional units that are part of the platform are expected to be present in similar form on all spacecraft systems. As these units have to be tightly integrated with each other, CubeSat developers are usually dependant on the platform of a single supplier. Integrating different platform modules is not easily achievable and requires considerable effort [SM23], which is why LibreCube aims to develop an open hardware and software system that provides a robust platform as a basis for CubeSat development.

On the *Payload* side, there are the mission-specific modules that host scientific hardware, such as sensors, cameras and actuators. While payload hardware is usually uniquely developed for the cause of the CubeSat mission, it can still benefit from a standardized interface to the system bus and processing unit towards robust inte-

gration and reusability.

In this section we will briefly go over the different functional units of the FRA, their definition according to SAVOIR [Eur21] and their implementation in our system.

Processing

The processing function is the central element of spacecraft avionics, responsible for data handling and executing the on-board application software, typically hosted on dedicated OBCs. It interfaces with multiple subsystems, communicating with ground by encoding and decoding telecommand and telemetry packets, and monitoring and commanding onboard equipment over the onboard bus. Its functionality includes general-purpose processing, volatile and non-volatile storage, boot and self-test mechanisms, automation, time management, error detection and correction. Redundancy is typically implemented for reliability, with cold redundancy preferred for power savings and warm redundancy employed in time-critical missions. Error detection relies primarily on internal mechanisms, such as watchdog timers, memory checks, and hardware voltage monitoring [Eur21].

Professional satellite systems use special space-qualified hardware to host their processing function with maximum reliability. As a notable example, ESA uses a dedicated processor architecture *LEON*, which features built-in error correction on registers and cache, redundant I/O communication interfaces for space protocols, and overall SEU-resistant and -tolerant design [AGW10]. Similarly, Error Detection and Correction (EDAC)-memory is used for both working memory and data storage [Eic12].

However, since CubeSats typically have lower reliability requirements, they often use standard microprocessors on commercial off-the-shelf OBC modules, which usually provide redundancy through a backup processors.

For this work, since the implementation is only intended to be used as a reference and for testing purposes, we decided to target a Raspberry Pi 5 single-board computer. It has the advantage of being readily available, and supported by a wide array of open source software, hardware accessories and instructions, due to its popularity. The hardware would be suitable to be used on a CubeSat as some missions have already shown [Uta], but further evaluation on the OBC hardware will be necessary for the final space-worthy system, as described in detail in Section 5.3.

The SAVOIR mapping suggests allocating the processing function to a dedicated *Processing Module* which is equivalent to an OBC, which we will also follow in our system.

Data Storage

The data storage functions describe the onboard memories that hold data collected onboard throughout the mission. The SAVOIR architecture differentiates between

Platform Data Storage and *Payload Data Storage*.

The platform data storage is specifically responsible for storing platform-related data, such as housekeeping reports when not connected to the ground, as well as other M&C-related files like automation scripts. Due to its critical role, it is subject to strict reliability requirements; however, it does not demand significant storage capacity or bandwidth, as the report data is relatively small [Eur21].

The payload data storage stores scientific data that was collected by the payload hardware. This memory typically requires high capacity and bandwidth to hold big datasets, such as high resolution images taken by onboard cameras. However, the reliability and data integrity constraints are less strict, as partial loss or corruption of science data does usually no endanger the success of a mission [Eur21].

On CubeSat missions, platform and payload data storage functions are typically both hosted on the same hardware for simplicity, such as a hard drive connected to the OBC, which matches with the SAVOIR CubeSat mapping. For our test system, the platform and payload data functions are both hosted on the SD-card that also serves as the Raspberry Pi's boot drive.

Safeguard Memory

The Safeguard Memory (SGM) stores critical contextual information for the OBSW, such as the spacecraft configuration, mission phase, and maneuver data. This allows the OBSW to correctly resume essential tasks following a restart caused by a switch-over to the redundant OBC. Communication with the SGM is handled via the onboard bus, and the module is specifically hardened against radiation-induced errors.

The SGM operates in a hot-redundant configuration, where both the nominal and redundant memory banks are accessible by both processing units. Any memory update must therefore be applied simultaneously to both memory instances to ensure consistency.

As indicated by the CubeSat system mapping, an SGM is typically not required for CubeSat missions, where manual reconfiguration is performed after an OBC switch-over. In the LibreCube hardware design, however, the SGM has been integrated on an optional *Reconfiguration Module*, using an EDAC-protected EEPROM.

Telecommand Decoding & Telemetry Encoding

The telecommand decoding and telemetry encoding functions are responsible for translating the lower data link layer protocols transmitted by the antenna to the higher level *CCSDS Frames* that are generated and consumed by the OBSW. The data is processed according to the CCSDS protocol chain (see Figure 2.2), which structures the data handling in layers inspired by the ISO OSI model [Eur21].

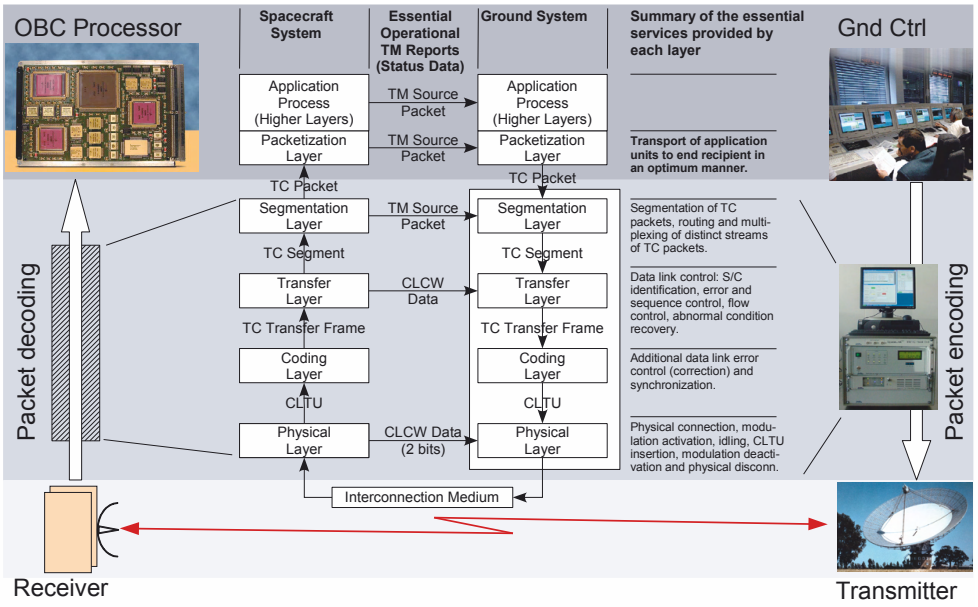


Figure 2.2.: *CCSDS TC encoding / decoding. Source: [Eic12]*

Notably, the standard splits the OSI Data Link Layer into three specialized sublayers:

- **Channel Coding and Synchronization Sub-layer:** Responsible for selecting the best quality signal modulation, suitable bandwidths, and multiplexing of multiple redundant transmitters. The two relevant protocols for this layer defined by the CCSDS are called Command Link Transfer Units (CLTUs) for TC uplink, and Channel Access Data Units (CADUs) for TM downlink.
- **Data Link:** Bundles data into containers, called *Frames*, which ensure correct routing between multiple spacecraft by a spacecraft ID, and also detect and correct sequencing errors. This layer also allocates data streams to *Virtual Channels*, which can prioritize data streams based on bandwidth restrictions on downlink, or target specific decoder hardware on uplink.
- **Segmentation:** Additional segmentation layer that splits up big data link frames and provides and additional routing with a *MAP ID* for targeting specific onboard TC consumers.

Due to its importance to offer recovery in the event of OBC failure, the telecommand decoding function has to be hot redundant, so that if one decoder fails, the other is still operational and can be targeted by mission control. Whereas the telemetry function can be cold redundant, as mission control can blindly send commands to switch to the redundant encoder in the event of a failure of the nominal encoder.

SAVOIR defines separate functions for *Platform Telemetry* and *Payload Telemetry*, as professional agency satellites often use dedicated high-speed data link for transmitting payload data to ground [Eur21]. Platform communication is usually transmitted in S-band (2 - 4 GHz), as it offers good tolerance over bad weather conditions. Pay-

load data links benefit from greater bandwidths, so often the X-band (8 - 12 GHz) or Ka-band (27 - 40 GHz) is used, which are however more vulnerable to weather conditions [Eic12].

The CubeSat mapping indicates that telecommand and telemetry transponders can be allocated to a communication module (COMMS), while the payload telemetry function is considered optional, as most CubeSats don't need the extra bandwidth and transmit payload data through the S-band. Physically, the TC/TM functions are usually implemented on dedicated microcontrollers, sometimes called CCSDS processors [Eur21], [Eic12].

The LibreCube COM module follows a similar approach: Two hot redundant transponder controllers operate in parallel and are both connected to the antenna through a diplexer, and are both connected to their own UART bus for data transmission to to the rest of the system. For monitoring and control purposes by the OBSW, the transponders are also connected to the SpaceCAN onboard bus [Lib].

Essential Telecommand, Telemetry & Reconfiguration

When a critical part of the avionics fails and can not recover on its own, mission control has to have some controllability over the system to reconfigure and restore its functionality. For this reason, satellites have *Essential Telecommand* functions, which are processed by decoding hardware before the OBC. The essential TC receives High Priority Command (HPC) sent by ground, which is routed on board to the essential TC processing function by an associated *MAP ID*.

The HPCs contains commands to reconfigure the spacecraft – for example it could command to power cycle the nominal OBC, or to boot up the redundant one. It acts by manipulating special platform commanding signal lines that have mission-specific meaning, it could for example enable or disable a relay that powers an OBC. Historically, the hardware that powers these commanding lines is called Command Pulse Decoding Unit (CPDU), as in the early days of spaceflight, electrical onboard reconfiguration was done with bistable mechanical relays which would be tripped by signal pulses, and they would store their configuration reliably by their mechanical position [Eic12]. Due to the importance of the essential TC function, it is required to operate in hot redundancy, and it is recommended to be implemented in hardware of low complexity. Typically in agency satellites it is integrated into the TC decoding hardware [Eur21].

To monitor a failing spacecraft, there has to be a reliable return channel of essential onboard parameters to ground that work independently of the OBC. The *Essential Telemetry* function collects critical onboard status parameters, such as the current hardware configuration, voltage levels, temperatures and software reports. This report, called High Priority Telemetry (HPTM), is then injected into the telemetry stream in a dedicated virtual channel [Eur21].

SAVOIR also defines a *Reconfiguration Function*, which monitors system alarms triggered by subsystem failures (e.g. a watchdog detects a crashed OBC), and can re-

configure the spacecraft autonomously to recover from failures [Eur21]. This level of autonomy is often disregarded on CubeSats, which is why the reconfiguration function is not included in the CubeSat mapping [SM23].

As we introduced the SAVOIR architecture to compare it against the LibreCube hardware as part of this work, we noticed that the essential telecommand, telemetry and reconfiguration functions were not yet reflected by the existing LibreCube modules, so we looked into possible integration options. The SAVOIR CubeSat mapping suggest to allocate the essential TC & TM to the COMMS module and to disregard the reconfiguration function. However, as some CubeSat missions disregard these functions entirely and instead rely on autonomous per-module recovery, we decided to keep the modular approach and introduce a new *Reconfiguration Module*. Apart from the essential TC & TM functions, this module hosts the CPDU, reconfiguration function, Safeguard Memory (SGM), and a reference clock. We describe the development of this module in Section 3.2.

Data Links

The data links interconnect the onboard hardware modules through communication bus systems. SAVOIR defines two distinct types of onboard data links:

- **Command & Control Data Link:** Connects all onboard hardware for transmission monitoring and control data, such as sensor parameters or actuator commands.
- **Mission Data Links:** Used for high-volume or auxiliary onboard data transfers, such as scientific payload data.

Agency satellites typically use a combination of MIL-STD-1553B, SpaceWire and CAN for both data link types [Eur21].

But for CubeSats, specialized hardware for space bus protocols such as MIL-STD-1553B or SpaceWire is not viable, so they typically use a combination of more easily available bus hardware such as I^2C, SPI, USB and UART [BLG17], with more recent developments towards CAN and Ethernet [Cho+20].

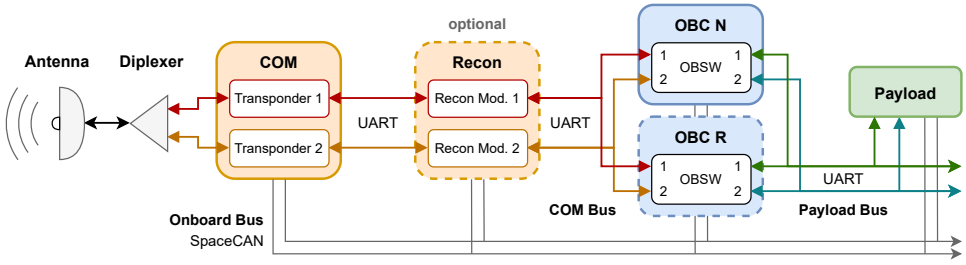


Figure 2.3.: *Bus Systems on the LibreCube Platform*
The two hot redundant communication chains to the antenna are shown in red and orange on the left side; the two redundant mission data links to the payload modules are shown in green and blue on the right side.

The bus communication concept implemented on the LibreCube hardware platform is illustrated in Figure 2.3.

We use a SpaceCAN bus as the command & control data link, and two redundant UART connections for mission data transmission. According to the SAVOIR recommendations, command & control data links should be operated in cold redundancy, with switch-over either detected and commanded from ground or by the OBSW [Eur21]. However, the LibreCube SpaceCAN bus operates intrinsically in warm redundancy, allowing the active bus to be switched either automatically – based on bus activity and detected errors – or manually by the OBSW[Sch+19].

In contrast, the operation of the mission data links is highly dependent on the specific mission payload. Consequently, we have not yet developed standardized access protocols; however, we have identified this topic as a potential area for future development.

For communication between the onboard computer and the COM module, we utilize two hot-redundant communication chains via UART. Both the nominal and redundant OBCs are connected to both COM chains. No special handling is required in this setup, as the OBC modules operate in cold redundancy – meaning that only one module is active at any given time.

Onboard Time

To orchestrate the different parts of the system synchronously, and to get a traceable view of onboard events, it is important to keep a consistent time distributed through the entire avionics system. The onboard time is managed by the *Onboard Time* function which is responsible for keeping and distributing the onboard time, and allow for synchronization with ground or a separate *Time Reference*. This time reference could be for example a high precision clock on board, or a receiver of other external reference time systems such as GPS.

For our system, the onboard time function is integrated into the OBSW. As we use a Raspberry Pi computer, we can use its built-in real time clock for time keeping, and the time synchronization methods of the *SpaceCAN* bus for time distribution through the system. We decided to also include a high precision clock as an experimental time reference as part of our *Reconfiguration Module*, which can be synchronized with the onboard time of the OBC.

2.3. Protocols and Interfaces

In this section we describe the relevant protocol and interface specifications that we implemented in our software stack.

Packet Utilization Standard (PUS)

The Packet Utilization Standard (PUS) standardizes the service-based monitoring & control aspect of the onboard software. It defines sets of functionality bundled into services and their associated application-layer protocol to monitor and command the spacecraft system from ground. For a mission specific implementation of PUS, it is not expected to fully implement all parts of it, but to taylor the implementation towards specific demands, and to extend it with desired functionality [Eur16].

The PUS architecture consists of the following parts [Eur16]:

- **Application Process:** The application process is responsible for hosting the services, and the encoding, decoding and routing of packets. It is uniquely identified by the *Application Process Identifier (APID).* A satellite system can host multiple such application services to distribute the M&C system, but for our implementation just one is sufficient.
- **Service:** A service defines a set of functionality and its required system interfaces. It is identified by the *Service Type ID*, with ID < 128 reserved for predefined services, and ID ≥ 128 free to be used by custom services. The latest revision C of PUS defines 20 of such predefined services which can be found in Appendix A.
- **Subservice:** A subservice is a part of a service that concretely defines the functions and the structure of the TC and TM packets used to for communication with ground. The packets defined by a subservice are called *Messages* and use predefined *Message Subtype IDs* to identify the type of message. The subdivision of services into subservices can be regarded more as an organizational tool rather than a technical function, as this distinction is not relevant for the implementation.

In a typical satellite scenario with one spacecraft and one ground-based mission control instance, there is one onboard application process and one ground application process which communicate with each other through TC and TM packets (see Figure 2.4). The onboard application process hosts subservice providers, which implement their predefined functionality, can be commanded with TC packets and report back with TM packets. The ground application process hosts subservice users, which generate the TC packets in order to request the desired onboard functionality, and interpret the responded TM packets and extract the report data [Eur16].

The packets used for communication by PUS are based on *Space Packets*: The Space Packet Protocol is an internationally recognized and widely adopted standard for data transmission in space applications, defined by the CCSDS [Con20b]. A Space Packet consists of a primary header, an optional secondary header, the data field and an optional error control field. The list of PUS packet header fields can be found in Appendix A.

As the data contained in a packet's data field typically consists of multiple concatenated values that must be serialized into bytes, the PUS standard specifies how this

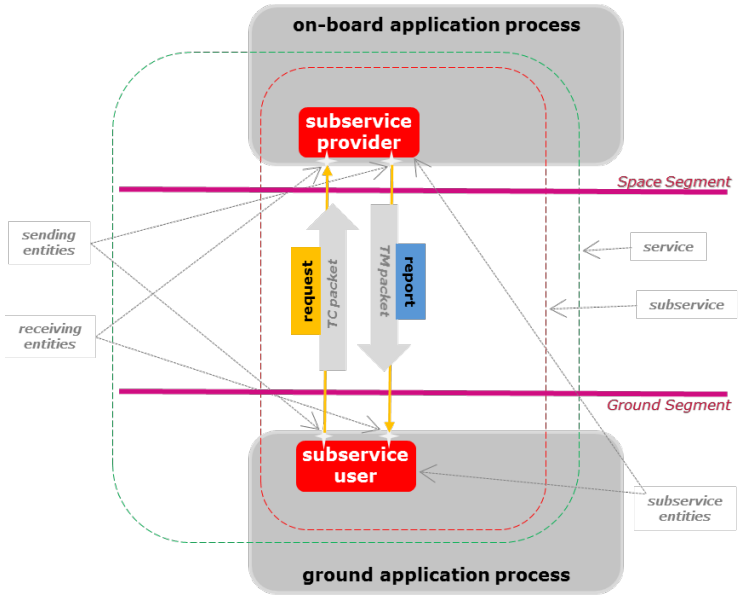


Figure 2.4.: *The space to ground PUS service system context. Source: [Eur16]*

serialization is to be performed. The specification defines the structure of data fields for specific service requests and responses, which include individual basic data type fields as well as more complex constructs such as lists and deduced structures. The serialization rules for basic data types – such as booleans, numbers, and strings – are described in detail using so-called *Packet Field Types*.

When a service receives a command, it provides feedback on the progress, success, or failure of the execution to the ground segment in the form of *Verification Reports*. These reports, transmitted as TM packets generated by Service 1 – *Request Verification*, reference the corresponding TC packet that initiated the request by including the unique packet count number of the TC packet's header. By correlating verification reports to previous requests, the ground segment can track and verify the correct execution of sent commands.

CCSDS File Delivery Protocol (CFDP)

Modern satellites collect and store data not anymore in tape recorders, but in digital files on mass memory. There is therefore the need to reliably exchange files between the spacecraft and the ground segment. Many satellite missions implement custom file transmission protocols, for example by utilizing PUS Service 13 – *Large Packet Transfer*, which in turn requires additional development and testing. For this reason the CCSDS developed a standardized file transmission protocol for space applications, the CCSDS File Delivery Protocol (CFDP) [Con20a].

CFDP defines a *Protocol Entity*, which describes a software module instance that handles the protocol and interfaces with the onboard software. This entity works on a virtual filestore, which is a mapping to a physical filestore on the hosting system,

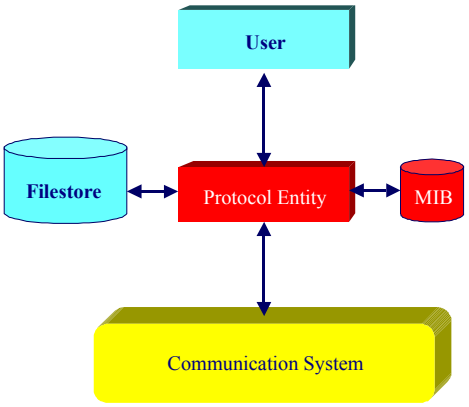


Figure 2.5.: *Architectural Elements of the File Delivery Protocol. Source: [Con20a]*

and which is exposed to CFDP file operations. The entity communicates to remote entities through a communication system. It can work with any packet based communication protocol, but for space applications typically Space Packets are used, which we will also use in this work. The protocol can be configured with parameters stored in a Management Information Base (MIB). CFDP entities get assigned a unique identifier, which may be mapped to any identifiers used by underlying protocols, and it does not differentiate between ground or space entities.

CFDP offers two modes of file transmission:

- **Unacknowledged:** The sender sends file data without any feedback from the receiver; the receiver may detect errors and discard the erroneous file, but no automatic retransmission is triggered.
- **Acknowledged:** The receiver sends Negative Acknowledgments (NAKs) to the sender if it detects any errors or missing file segments in order to trigger the retransmission of the affected segments.

Apart from file transmission, CFDP offers other filesystem operations such as renaming, deletion and creation of files and directories on remote entities. However, this functionality overlaps with the PUS Service 23 – *File Management*, which we decided to prefer over the CFDP filesystem operations. One interesting quirk of CFDP is that it only offers a *Put* request to initiate an upload towards a remote entity, but no *Get* request to initiate a download from a remote entity. This functionality is instead covered by *Proxy Operations*, which allow one entity to command another entity to perform an operation. In order to download a remote file, the local entity sends a proxy operation request to the remote entity to perform a *Put* operation of a desired file towards the local entity [Con20a].

As both, the PUS protocol and the CFDP protocol communicate through Space Packets in our system, the individual packets have to be routed to their designated process. This is done with the APID field in the Space Packet header (see Appendix A), with the PUS process and CFDP entity both having individual APIDs.

SpaceCAN

SpaceCAN was developed by Artur Scholz et al. in 2019 as an extension of the CAN bus protocol, with the aim of adapting it to the specific requirements of onboard communication systems while avoiding unnecessary complexity [Sch+19]. The protocol uses two CAN buses in warm redundancy, meaning that communication is performed on one active bus, while the other bus is on stand-by and can be switched to in case of a communication problem on the active bus.

The CAN bus interconnects multiple nodes, all of which are capable of broadcasting and receiving messages transmitted over the bus. To prevent access conflicts, the protocol employs a message arbitration mechanism based on message identifiers: lower numerical values correspond to higher priorities. If two nodes attempt to transmit simultaneously, arbitration ensures that the node with the lower-priority message (i.e., the higher ID value) ceases transmission as soon as it detects that one of its identifier bits is overwritten by a dominant bit (logical zero) from a higher-priority message. CAN incorporates mechanisms for reliable data transmission: a message includes a checksum field to check data integrity, and in case a node receives an erroneous message, a NAK error frame is sent to initiate a retransmisssion.

Topology-wise, SpaceCAN follows a master/slave architecture, including exactly one master node and multiple responder (slave) nodes. The master is responsible for sending commands to the responder nodes, which in turn report responses and status updates back to the master.

SpaceCAN overlays a communication protocol on top of the CAN standard, using CAN message identifiers to encode protocol-specific information. It includes mechanisms for addressing nodes individually and transmitting fragmented data across multiple CAN messages, thereby overcoming the eight byte size limitation of standard CAN frames.

In addition to basic command/response communication, SpaceCAN supports optional service-based interactions inspired by the PUS protocol, offering functionalities such as parameter reporting and request verification.

To ensure system-wide time synchronization, the master node periodically transmits dedicated messages containing the current system time. Furthermore, it sends regular *Heartbeat* messages, which are monitored by the responder nodes to determine the currently active communication bus. If heartbeat messages are not received for an extended period, the responder nodes automatically switch to an alternative bus to maintain connectivity and redundancy.

Procedure Language for Users in Test and Operations (PLUTO)

The Procedure Language for Users in Test and Operations (PLUTO) language provides several domain-specific features that are particularly well-suited for spacecraft monitoring and control tasks [Eur08]. Spacecraft are highly sensitive systems in which incorrect commands may result in erroneous configurations. Such faults

could necessitate complex recovery procedures and, in the worst case, lead to mission failure. Therefore, any automation implemented within the spacecraft system must ensure that the correct execution and response of the avionics system to automated actions are verified. To this end, PLUTO includes mechanisms to confirm the expected system state before, during, and after the execution of automation scripts.

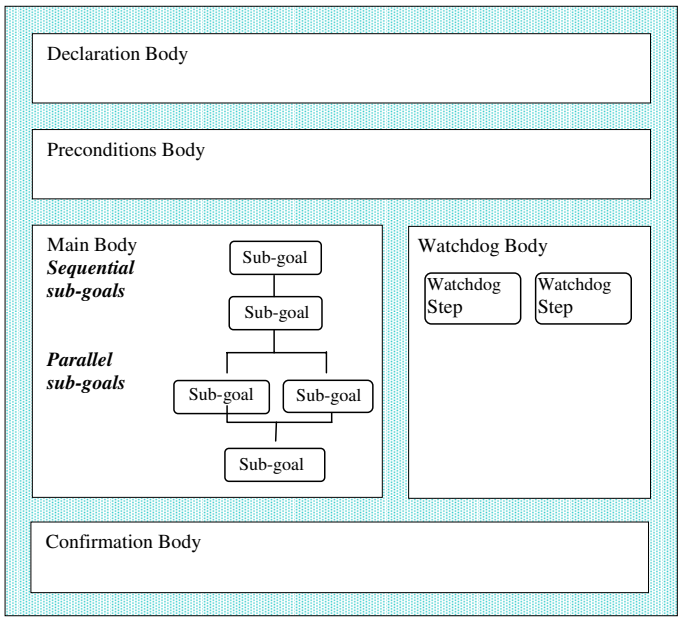


Figure 2.6.: *PLUTO procedure and its elements. Source: [Eur08]*

A PLUTO script defines a *Procedure*, which resembles a program that can be invoked and managed by an execution engine. Such a Procedure consists of multiple distinct body blocks – as illustrated in Figure 2.6 – that specify various aspects of its behavior, although only the main body is mandatory.

The first block is the *Declaration Body*, in which custom local events can be defined. These events can be raised to trigger contingency actions, which are handled immediately and meanwhile pause the procedure execution similar to system interrupts.

In the next block, the *Preconditions Body* defines the conditions that must be satisfied prior to execution. These preconditions may, for instance, specify that certain parameters must lie within defined limits or that a device is set to a required configuration. The precondition declarations allow to either wait until the precondition is met, or to immediately reject the launch of the procedure.

The logic of the procedure is implemented in the procedure's *Main Body*. It consists of statements that are, by default, executed sequentially. However, the procedure may also include blocks of statements that are executed in parallel. Such a procedure statement could involve initiating an operation of the onboard system or to execute a *Step*.

A *Step* is a chunk of work to be executed within a procedure and is defined similarly

to a procedure, with the key difference that statements within a step may include control flow logic, which is not permitted directly in the main body of a procedure. This design choice was made to clearly distinguish the sub-goals a procedure aims to achieve, while the implementation details of how to reach these goals are contained within the individual step definitions.

A procedure or step can additionally define a *Watchdog Body*, defining contingency conditions to monitor during the execution of the *Main Body*. If a contingency condition is triggered, the execution of the main body is paused and further steps to resolve the situation or to abort execution are defined in further statements in the watchdog body.

Finally, a procedure or step can define a *Confirmation Body*, which checks conditions to confirm if the execution was successful. If the confirmation conditions were not satisfied, the procedure or step returns a "`not confirmed`" status. An "`initiate and confirm`" statement to execute a step can be followed by a continuation test, which defines if non-confirmed step should be restarted, or if the procedure should resume or abort execution.

PLUTO offers features that are typically found in scripting languages, such as local variables, basic data types, logging and flow control statements such as `if/else` branching and `while/for` loops. To interface with the spacecraft system, PLUTO includes statements to read onboard parameters, launch onboard operations and react to onboard events.

2.4. Onboard Software Architecture Overview

The onboard software serves multiple purposes. It hosts software which controls vital automations of the satellite system in order to gather scientific data, perform maneuvers, and to keep the system working. But it also provides access for Monitoring and Control (M&C) from the ground segment. The SAVOIR OSRA specifies a reference architecture with all the functional parts for such an onboard software. While traditional CubeSats typically do not conform to established space agency standards – often relying instead on custom software architectures or specialized CubeSat frameworks – this work aims to demonstrate the feasibility of applying open European standards for satellite technology to CubeSats. To that end, we present an implementation that closely follows the OSRA architecture, which we specifically tailored for CubeSat platforms. In this section, we briefly introduce the key aspects and functional units of the SAVOIR OSRA, and how they transfer to our implementation, which we discuss in detail in Section 3.1.

As shown in Figure 2.7, the OSRA architecture is structured into three layers: the *Component Layer*, the *Interaction Layer*, and the *Execution Platform*.

The **Component Layer** contains applications that perform tasks highly dependent on the specific mission profile and the spacecraft's hardware. For instance, this includes software responsible for controlling the satellite's attitude in orbit (the AOCS)

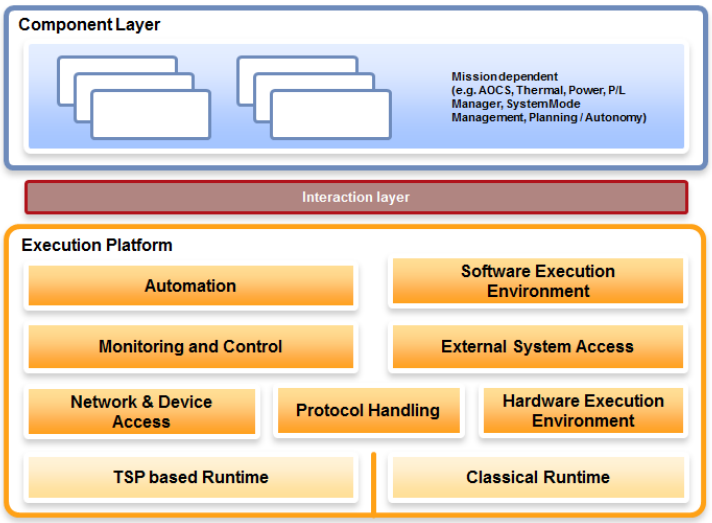


Figure 2.7.: *OSRA Three-Layer Architecture. Source: [Eur21]*

or a state machine that reconfigures the spacecraft through various mission phases. These applications require access to the entire spacecraft system in order to monitor and control hardware components and to collect scientific data.

To support the development and verification of such applications, and to promote software reusability, the OSRA introduces an abstraction of system access through an **Interaction Layer**. Specifically, the OSRA suggests to structure the onboard software into *Components*, which define a component interface made of *Attributes* and *Operations*, which then communicate with each other through connectors as part of the interaction layer. These connectors are recommended to be tool-generated, based on predefined component connections, which makes sense for thoroughly planned and tested agency missions. However, as our reference implementation is intended to be used as an easily modifiable platform for testing purposes, such a rigid component structure would hinder agile development processes in the prototype phase of a CubeSat system's development as stated in Concern CON 3 – *Testing Platform*. To this end, we decided to implement the interaction layer as a software bus on which components can independently connect to each other without the need of additional configuration, as described in detail in Section 3.1.1.

Finally the third layer – the **Execution Platform** – hosts basic spacecraft software functionality required for reliable software execution, system access, and monitoring and control from ground. The OSRA groups the various functions of the execution platform into distinct functional groups, which are briefly introduced in the following. While we outline our implementation approaches for these functions here, a detailed discussion of the design rationale and implementation specifics is provided in Section 3.1.

Runtime: The runtime refers to the operating system on which the onboard software stack executes. It includes basic device drivers, a file system, and provides mechanisms for concurrent task execution. As an alternative to a classical runtime,

the OSRA defines the option of employing a Time and Space Partitioned (TSP) system, which enforces strict partitioning of hardware resources – such as processor runtime and memory space – on a per-task basis.

For our implementation, we opted to use a Linux kernel, which uses concurrent software processes and includes drivers for the CAN and UART interfaces required for our system.

Network & Device Access: This group is responsible for communication with onboard equipment and data acquisition via the onboard bus systems. The CCSDS has developed a specialized framework for this purpose – the Spacecraft Onboard Interface Services (SOIS) [Con13]. SOIS is a comprehensive framework including multiple communication layers and protocols, designed to support a wide variety of hardware and bus systems in a modular and reusable manner. While this approach is well-suited for professionally developed satellite systems by space agencies, it represents an unnecessary level of complexity for CubeSat-class missions. For this reason, we adopt a simplified alternative in our implementation. In our approach, each onboard device is represented by an *Equipment Component* that manages communication with the respective hardware and exposes its functionality to the interaction layer via a component interface. Since our system relies on SpaceCAN as the onboard system bus, the communication handled by the equipment component is reduced to generalized SpaceCAN service interactions. To support this design, our software stack includes a dedicated SpaceCAN driver module – the *SpaceCAN Master*. This module manages the underlying CAN communication and provides a simplified API for performing SpaceCAN operations.

Protocol Handling: This group provides the functionality required to manage the communication protocols with the ground segment for Monitoring and Control (M&C) and file transfer, as well as for packing and unpacking the corresponding data packets into and from data link protocols. In our software stack, we employ the PUS protocol for M&C, CFDP for file transfer, and CCSDS Frames as the data link protocol. Our stack includes a *Frame Processor* module, which unpacks the frames received from the COM module and routes the contained packets to their respective destination processes. Subsequently, the PUS and CFDP packets are processed by their dedicated modules.

Hardware Execution Environment: This group provides core infrastructure services essential for the operation of onboard software, including monitoring of the OBC and its OS, time management, and data storage. Access to the onboard time is provided, and its distribution across the system is ensured via the onboard bus infrastructure; this is realized by using the Linux kernel's system clock in combination with the time synchronization services of SpaceCAN. Additionally, this functional group includes access to an onboard file system used by internal software components and the file transfer protocol, which is provided by the Linux kernel in our implementation.

Monitoring and Control: The Monitoring and Control (M&C) aspect is a vital part of the onboard software, as it provides functionality to service and command the

spacecraft from the ground segment. Notably, the M&C system includes functions to access onboard parameters (voltages, temperatures, etc.), reporting of onboard events (errors, out-of-limits, etc.), and commanding. As the OSRA suggests, we are using our own PUS implementation as our M&C system, which defines dedicated services for the mentioned functions. In particular the parameter access and reporting is provided by Service 3 – *Housekeeping* and Service 20 – *Parameter Management*, event reporting is provided by Service 5 – *Event Reporting*, and commanding is provided by custom services.

External System Access: In order to allow advanced, low-level system maintenance beyond dedicated service interactions, the system has to be accessible at a low level from ground. This is provided by PUS Service 2 – *Device Access*, which allows raw byte-level communication with onboard equipment.

Automation: Satellites are typically not always connected to a ground stations which can execute commands whenever desired in order to react to onboard events or to reconfigure at predefined points in the mission. Therefore, the onboard software must provide mechanisms for autonomous operation. In this context, the PUS standard includes several services designed to support automation. PUS Service 11 – *Time-Based Scheduling*, Service 22 – *Position-Based Scheduling*, and Service 19 – *Event-Action* offer fundamental capabilities for implementing reactive automation onboard the spacecraft. For more sophisticated automation tasks, an Onboard Control Procedure (OBCP) engine is employed. This engine allows interaction with onboard systems through automation scripts that support advanced control flow logic. To facilitate the management of such scripts, PUS Service 18 – *On-Board Control Procedure* provides functionalities to upload and manage OBCP scripts, and to control the execution engine. For our system, we implemented an OBCP engine capable of executing PLUTO scripts.

Software Execution Environment: The software execution environment encompasses various supporting functionality for the software stack. This includes context management, error reporting, support libraries and life-cycle management. The context management describes the software's capability to store and recall vital state information in non-volatile memory in order to recover from a restart. Our PUS implementation incorporates mechanisms for services to store and recall context data in form of atomic file operations, which can be used in a similar fashion by custom application software. The life-cycle management is responsible for initializing the onboard software on system start-up, monitoring it for faults, and restarting any components that crash. In our current implementation, a simple starter script is used to bring up all system components, although further development is planned to enable launching and monitoring each component in independent processes.

3. Main Contributions

3.1. Onboard Software Implementation

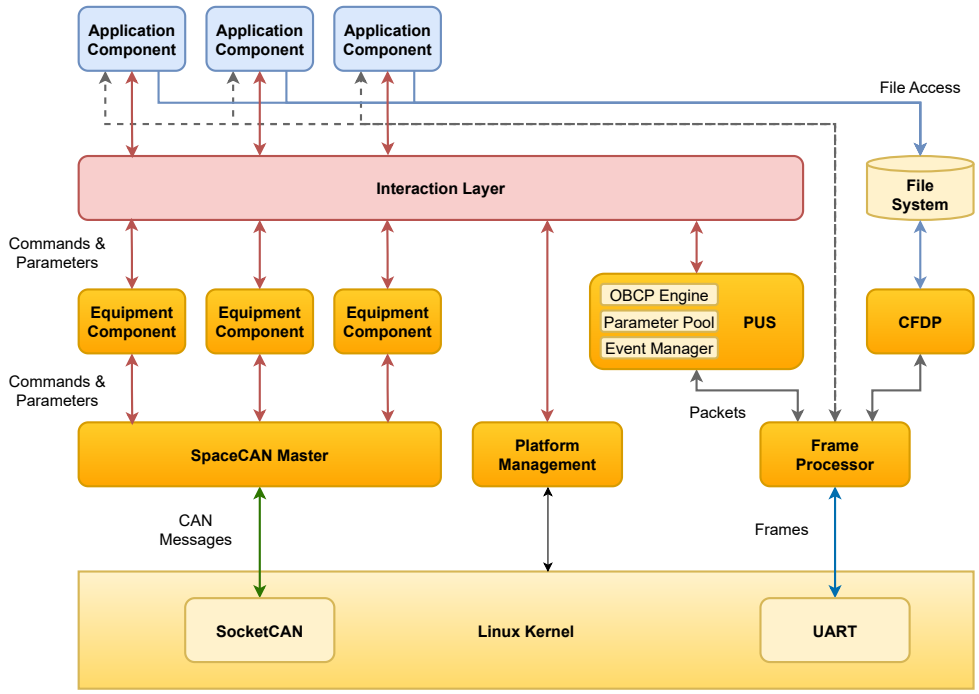


Figure 3.1.: *Structural Overview of our Software Stack*

For the software stack, we designed an architecture that fulfills the requirements outlined in Section 2.1, while also adhering closely to the recommendations of the SAVOIR OSRA. In fact, nearly all functional components defined by the OSRA can be directly mapped to corresponding functional units within our software stack, as demonstrated in Section 2.4. The overall structure of our software stack is illustrated in Figure 3.1, which we will elaborate in this section.

Similar to the OSRA, our software stack can be divided into three layers:

- **Application Layer:** This layer includes the mission specific applications that control the onboard equipment and collect scientific data according to the mission profile. Our software provides all the required interfaces for custom application components to interact with the spacecraft system.
- **Interaction Layer:** This layer acts as a software communication bus to connect the various component processes. It offers a generalized way to access

parameters, events, and commands from one process to another. OSRA proposes connectors between specific components, but we chose to implement an interaction layer that allows any component to communicate to any other component.

- **Execution Platform:** This layer provides basic platform functionality required by the application components, as well as monitoring and control capabilities for ground control. It includes the runtime environment – specifically, a Linux kernel in our case – along with components responsible for handling communication protocols such as SpaceCAN, CCSDS frames over UART, and CFDP. Moreover, it integrates the PUS implementation, including all its services and sub-components. In our architecture, this layer additionally incorporates equipment components that manage equipment-specific device access protocols, making them accessible to the interaction layer. It also comprises a platform management component, which is responsible for starting and monitoring the software processes and the OBC hardware platform.

In the following subsections, we discuss the purpose and implementation details of each component of our software stack.

Runtime

The runtime provides drivers to access the OBC hardware, and also provides an operating system that can run and manage multiple software tasks concurrently. The OSRA suggests either of two options, a Time and Space Partitioned (TSP) system or a classical runtime. A TSP system operates by strictly allocating system resources into distinct partitions for each application. This partitioning encompasses both memory space and processor time, thereby ensuring maximum isolation with minimal inter-partition interfaces. Such an architecture significantly aids verification and validation processes during the development phase [Eur21]. The other option of a classical runtime includes does not impose such restrictions and includes typical embedded operating systems such as Linux or FreeRTOS.

Due to our requirement REQ 7 – *Use of the Python Language*, we were limited in the selection of our operating system. To run Python on embedded hardware, there are essentially only the two options of using MicroPython or a Linux kernel. Both options offer the required functionality of hardware drivers, file system and task switching. But we opted to use a Linux kernel, as it offers greater flexibility and the option to exchange parts of the system with implementations in a different language than Python which better satisfies the requirement REQ 5 – *Customizeability*.

As our software only serves as a reference implementation, we simply chose to use the Raspbian operating system, as it integrates a solid driver foundation for the hardware of the Raspberry Pi computer, and various customization modifications are well documented due to its popularity. However for future developments, a real-time capable distribution could be considered.

Platform Management

This component is derived from the OSRA functional blocks *Hardware Execution Environment* and *Software Execution Environment*. It is responsible for initializing the software stack during start-up and for monitoring the OBC hardware, the operating system, and the running software processes.

In the current version of our implementation, the platform management component is realized as a simple start-up script, which initializes and configures all software components. More advanced functionalities have been deferred, as implementing automatic recovery mechanisms requires the integration of a potentially mission-specific FDIR concept. For basic prototype development, the current implementation is sufficient; however, we plan to extend the platform management component's functionality as needed.

A detailed insight into the starter script of our integrated system test is provided in Section 4.1.

Frame Processor

The frame processor serves as the interface between the onboard software and the antenna module for communication with the ground segment. In our system, it is responsible for decoding CCSDS frames and routing the contained Space Packets to the appropriate software processes based on their APID. Similarly, it collects packets from these processes, encodes them into frames, and transmits them to the antenna module.

To interface with the Communications (COM) bus connected to the antenna module, the frame processor transmits and receives frames via two hot-redundant UART connections. When a frame is received on either channel, its data integrity is verified, and its identifier is marked as received. Given the hot-redundant configuration of the COM connections, it is expected that each frame is received twice; the frame processor therefore discards any duplicate frames. The frame is then decoded using the CCSDS TC Space Data Link Protocol [Con21a], and the contained Space Packets are forwarded to the respective application processes via UDP sockets. To determine the UDP port corresponding to a given Space Packet's APID, we apply the simple formula $Port = 5000 + APID$.

Similarly, the frame processor listens on a dedicated UDP port for incoming TM packets, which are then bundled into TM frames using the CCSDS TM Space Data Link Protocol [Con21b]. These TM frames are subsequently transmitted over both COM UART buses.

The frame processor is currently still under development. Nevertheless, the system can already be used for testing by sending packets directly from the ground segment process to the UDP ports of the onboard software processes.

3.1.1. Interaction Layer

The interaction layer is an integral part of our architecture as it connects the different software components together. Therefore, it was of utmost importance to develop a robust and flexible implementation, which preferably can be implemented similarly on other operating systems, such as FreeRTOS. According to the OSRA, the interaction layer must support the inter-component-access of component interfaces which are comprised of *Attributes*, *Operations*, *Events* and *Datasets* [Eur21].

Attributes are parameters of a defined type, which can be either read-only as a *Data Attribute*, or read-write as a *Configuration Attribute*. The parameter access is typically implemented with *get* and *set* accessor functions. In our implementation we chose to not only support these accessor functions, but additionally support a publish-subscribe pattern, which allows for faster parameter update distribution compared to polling, and also eases the development of reactive software components. Component attributes are typically mapped to M&C parameters to be accessible by ground control. This mapping is configured in a so called *Spacecraft Database*.

Operations are functions that can be called with a defined set of parameters, with direction *in*, *in out* or *out*. An operation can be defined as synchronous where the call is blocking and either returns a success or an error, or asynchronously where the caller can resume its operation and gets notified over the execution status through status provision messages. For our implementation, as python does not support multiple function return parameters, we chose to restrict operations to return only one output parameter, which can however be a complex type with multiple fields. Due to requirement REQ 9 – *Convention Compliance*, we chose to not support *in out* function parameters, as they are generally regarded as bad practice [Mar08].

In M&C with PUS, there is no standardized way to call component operations from ground. Instead, the usual approach to expose operations to PUS is to define custom services with custom message types for each operation. As the definition and implementation of these custom services could be derived from the operation's function signature, this process could be automated by a specialized tool. However, this tool was not developed as part of this work and can be considered as a direction for future work.

Events are small sets of data that components can emit to connected components. Since our implementation does not rely on specific connections, we decided to emit events using a publish-subscribe pattern. Similar to events, the OSRA specifies that components can emit large datasets. In our implementation, we chose not to distinguish between events and datasets; instead, we allow events to carry large amounts of data. As with parameter mapping, component events are typically mapped to M&C events in the spacecraft database.

Communication Framework

For the implementation of the interaction layer in our software stack, we had to choose a suitable Interprocess Communication (IPC) framework to serve as a communication basis. It must support both the request-response pattern for synchronous function calls, and the publish-subscribe pattern for event emissions. For a Linux system, there are multiple solutions that we considered:

- **D-Bus:** The D-Bus is software bus used by the Linux system for interprocess communication [fre]. It allows the transmission of predefined message structures with both request-response and publish-subscribe pattern. However, implementations of the D-Bus are reportedly difficult to work with, which contributes to a lack of popularity and supporting documentation [Bes18].
- **MQTT:** MQTT is a messaging protocol developed for machine-to-machine networks, prominently used in IoT applications. In an MQTT network, a node publishes its parameter states, which are grouped into *Topics*. Other nodes can subscribe to these topics, but they can also publish to these same parameters, which could be interpreted as *set*-operations or a function calls. The protocol at its core works using the publish-subscribe pattern, but the latest version 5 also supports request-response patterns by using dedicated response topics and correlation data [Org19]. As result, MQTT appears to be a suitable candidate for our purpose, even if it is not typically used as an IPC mechanism.
- **Python Remote Objects:** The remote python objects of the *Pyro* library [Jon21] offer a simple way to expose python objects to other processes through network connections. An object exposed with Pyro is mirrored to proxy objects in the remote processes, which transmit operations performed on it to the exposed object, and receive result values in return. This would allow for a very simple implementation of the interaction layer, as components could simply expose their interface as a remote object. However, our experiments with Pyro encountered the issue that remote objects do not natively support the registration of callback functions in a straightforward manner, which is required for implementing the publish-subscribe pattern. Additionally, the communication mechanisms would be entirely hidden in the external library, which defeats the purpose of our work to serve as a reference implementation.
- **Native Network Sockets:** Communication with network sockets such as TCP and UDP is a very popular approach to IPC. It allows for full flexibility by implementing custom protocols, which however requires the development of such a protocols, including a form of data serialization. To communicate with native sockets, the process has to manually establish and monitor connections to sockets of other processes, and the extraction of message structures from a raw byte stream in case of TCP or multiple packages in case of UDP. This makes native network sockets difficult to work with, which is why we tried to avoid this approach.
- **ZeroMQ:** ZeroMQ is an asynchronous messaging library, which serves as a middleware to abstract the complexity of native sockets [Hin]. It offers simplified socket communication with messaging queues, that can be asyn-

chronously processed by the software process. The library handles the connections and data segmentation of the underlying native sockets, to allow for the transmission and processing of messages with large data capacity, and optional partitioning into multiple parts. ZeroMQ has specialized sockets for different messaging patterns, such as request-response and publish-subscribe.

Based on our evaluation, we identified MQTT and ZeroMQ as the most viable candidates for our implementation. Ultimately, ZeroMQ was selected, as its message queue mechanism closely resembles the inter-task communication using *xQueues* in FreeRTOS. Consequently, the implementation with ZeroMQ can serve as a reference for designing an interaction layer on FreeRTOS, satisfying REQ 8 – *Use of Universal Features*.

IL Implementation

We divided our implementation of the interaction layer into two parts, a *Transport* module which handles message transmission and the overlaying *Interaction Layer* module, which uses this transport and provides a simple API for the application code, as can be seen on Figure 3.2.

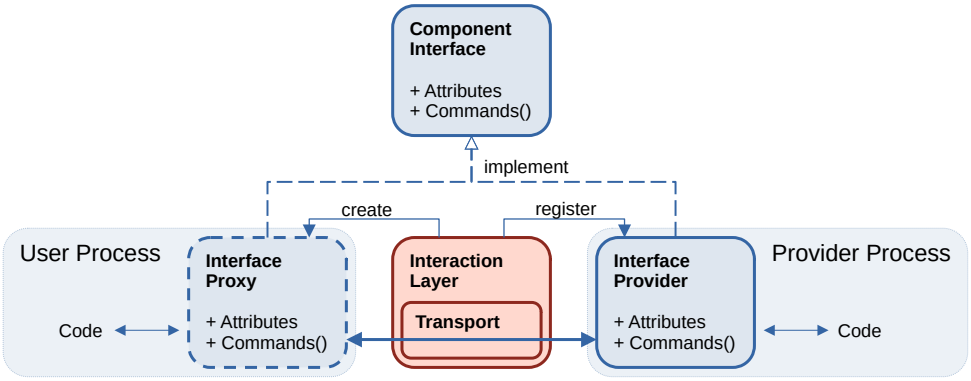


Figure 3.2.: *Interaction Layer Architecture Overview*

We took inspiration from the Pyro library, and implemented the support for proxy objects for simple remote access. A software component has to define an abstract *Component Interface* class, which includes attribute definitions, operation function stubs and event emitter definitions. This component interface should be part of a system-wide accessible library, so that any process could use this interface to create a proxy object.

The functional implementation of a component interface has to be provided by exactly one process of the system, which we will call *Interface Provider*. The component registers its interface provider to its interaction layer instance to expose it for remote access. Other tasks which want to access this component request the creation of a proxy object on their interaction layer instance with a reference to the abstract interface class. The interaction layer module then automatically builds an instance of this interface class with proxy attributes and functions, which redirect

interactions to the transport and ultimately to the interface provider. The creation and usage of proxy instances is however optional, and a user process could directly call the remote function using the transport module.

The transport module must support the following features:

1. Send function calls and receive its result
2. Receive function calls, call the corresponding application code and send the result
3. Subscribe to attributes and events
4. Publish attribute updates and events
5. Receive attribute updates and events and notify the application code

We designed our implementation in a modular fashion, so that as long as a transport implementation supports the mentioned features, it can be used with our interaction layer. However as mentioned previously, we decided to implement the transport with ZeroMQ. As the transport requires both request-response and the publish-subscribe pattern, the interaction layer has to create a *Response*-socket and a *Publish*-socket for each interface provider. To interact with these sockets, the interaction layer instance which hosts interface proxies on a user process creates a *Request*-socket and a *Subscribe*-socket (see Figure 3.3)

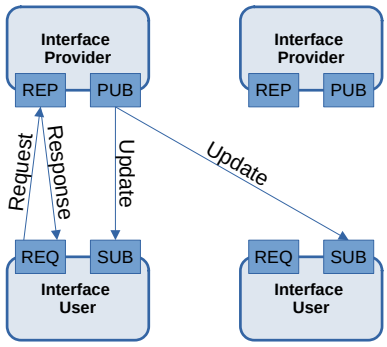


Figure 3.3.: *Component Sockets with ZeroMQ*

For the implementation of the required functionality of the transport, we developed a simple communication protocol using multipart messages transmitted through the ZeroMQ sockets. A request message is split into 2 parts: the function name to call, consisting of the component name concatenated with the function name (e.g. `SolarPanel.set_angles`), and a list of arguments (e.g. `[42, 29]`). The function name can be either the name of an component interface operation, or an access function of an attribute. A response message has 3 parts: the function name, a return code which indicates either success or an error, and a data field which contains the return value in case of success, or additional error data in case of an error. A publish message is similarly defined: its first part is the name of the publishing attribute, event, or asynchronous operation, followed by the published value.

The transport instance is responsible for the generation and interpretation of these

messages and to route the interaction between the designated component interfaces.

While ZeroMQ handles the segmentation and transport of messages via the underlying network protocol, the application still has to take care of the data serialization and deserialization from and to byte strings. For our implementation we decided to use JSON for this purpose, as it is widely supported in different software frameworks, is human readable, and requires no specific configuration, which simplifies development and especially verification and validation processes. JSON supports serialization of the basic data types *String*, *Number* (integer and floating point), *Boolean* and *Null*, as well as more complex structures such as *Arrays* and string-indexed dictionary structures known as *Objects*.

For our purposes, the provided data types are largely sufficient. However, we encountered two notable limitations: raw bytes (e.g., CAN messages) must be manually converted into character strings to be transmitted via JSON; and the *object* data type only supports string-based keys, which prevents the application from transmitting Python dictionaries that use other key types, such as integers. These problems can be avoided in practice, but may justify a directive for future work in order to find a replacement.

With ZeroMQ, a request message has to be sent to a specified response socket at a specified port. When a user process wants to interact with a component, it calls the request function of its interaction layer instance with the name of the desired component concatenated with the name of the desired function. The transport instance then has to determine the port of the response port of the component's interface provider to send the request message to. We implemented this name-to-port lookup as a dictionary included in the spacecraft database. This approach is functional but not flexible, as the all component interface providers require a predefined static port mapping. An alternative approach would be to implement the *Service-Oriented Reliable Queuing (Majordomo Pattern)* as described in the ZeroMQ guide [Hin], which adds a broker instance that automatically discovers service providers and routes requests based on service names.

While ZeroMQ automatically handles reconnections when a process restarts, a user process still has to manually handle timeouts when a socket does not respond. A non-responding socket would indicate a hung component. As the handling of this problem is dependant on the mission defined Failure Detection, Isolation and Recovery (FDIR) concept, we decided to only implement a simple finite retry approach, as described in the ZeroMQ guide as *Basic Reliable Queuing (Simple Pirate Pattern)* [Hin]. An approach for a more advanced handling of interaction layer timeouts would be to report the non-responding component to the platform management component or a dedicated FDIR application, which could for example restart the component process, the OBSW, the OBC, or command a switch to the redundant OBC.

An additional property of the ZeroMQ implementation must be taken into account: When a socket receives a request, this request must be fully handled before the next one can be processed. Consequently, a single socket does not support concurrent

request handling but instead enforces strictly sequential processing. This leads to the hazard of a deadlock: When a Process A requests Process B, and during the handling of this request, Process B in turn requests Process A, they will both be stuck waiting for the response of each other. However, in practice this hazard can be avoided with static analysis of the interactive structure of the system over the interaction layer. If a request has the potential to trigger a cycle of synchronous requests, such a cycle must be broken. This can be achieved either by converting one of the involved operations into an asynchronous call or by reassigning functional responsibilities to eliminate the cycle altogether.

3.1.2. Monitoring and Control with PUS

In accordance with Requirement REQ 11 – *Monitoring & Control with PUS*, we use Packet Utilization Standard (PUS) [Eur16] as our Monitoring and Control (M&C) system. We found one open-source Python-based implementation of PUS during our research – the *puslib* [pxn24] – but in order to provide an optimal integration with our interaction layer, we decided against a custom tailoring of this library and decided to develop an own implementation.

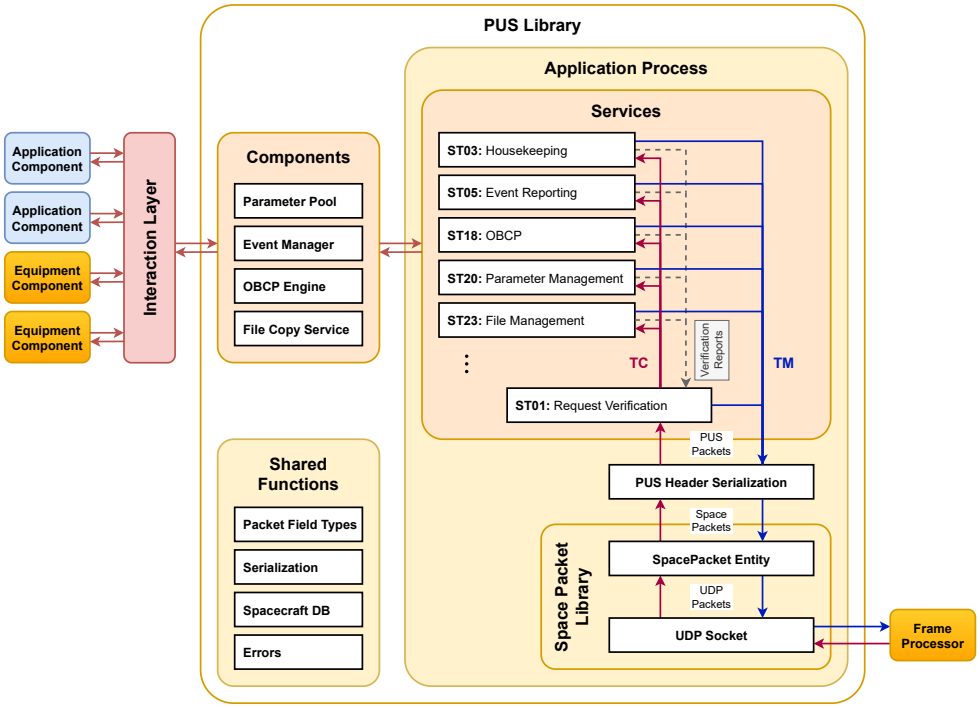


Figure 3.4.: *Structural Overview of our PUS Implementation*

The structure of our PUS implementation is illustrated in Figure 3.4, of which we will elaborate key elements in the following.

Application Process

The application process hosts the services and is responsible for serializing the secondary PUS header contained within the underlying Space Packets. It also incorporates a communication interface with the frame processor, which serves as a gateway for packet exchange with the ground segment.

The primary function of the application process is to manage packet communication. Since the services predominantly rely only on the data field of the packets, the handling of headers and routing is delegated to the application process. To receive and transmit packets to and from the ground segment, the application process communicates with the frame processor module via UDP sockets. The communication of Space Packets over UDP is handled by the *spacepacket* library, developed by LibreCube [Lib24c], which provides a simple API through a *Space Packet Protocol Entity*.

To support our PUS implementation, we developed *PUS Packet* classes for both TC and TM packets. These classes extend the Space Packet class by including the relevant secondary header fields (see Appendix A) and by providing dedicated encoding and decoding functionalities. Furthermore, the PUS packet decoder verifies the integrity of received packets using the checksum algorithms defined by the PUS standard.

In a mission tailored startup script, service instances are created and configured, including mission-specific custom services. These service instances are then registered on an application process instance, which aggregates the service instances in a lookup table. When the application process receives a TC packet, it is routed to the appropriate service to process it. According to the PUS specification, the task of forwarding these packets to the correct service is handled by Service 1 — *Request Verification*. Therefore, the application process passes all received TC packets to this mandatory service. To send TM packets back to the ground, the application process provides a function that services can call by passing the data field, service ID, and message type ID. The application process then generates the necessary packet header fields and forwards the completed Space Packet to the Space Packet Protocol Entity.

Some services require continuous cyclic processing, such as Service 3 – *Housekeeping*, which is responsible for the ongoing recording of parameter values and the autonomous transmission of reports. To support this, our application process incorporates an activity loop that executes the services' cyclic loop functions concurrently at a predefined interval, using Python's built-in *asyncio* library. Since there is a risk of system lock-up if a service fails to complete its loop execution within the allowed cycle time, the application process includes a safeguard: it checks in each cycle whether a service is still executing its previous loop. If this is the case, the process forcefully terminates the overdue execution. Such a forced termination may lead to an inconsistent system state, and therefore must be reported as an error. This allows either FDIR automation or the ground segment to take appropriate corrective actions.

Serialization

An important aspect of the PUS standard is the specification of reliable and unified transmission of telecommands and telemetry data. To this end, PUS defines the exact serialization of data values to be transmitted as bytes in the data field of the PUS packets. The data field of a PUS packet is made up of predefined message structures based on the message type, distinguished by the message subtype ID. This structure consists of multiple fields of specified data types, called *Packet Field Types*, such as booleans, integers, floating-point numbers, strings and timestamps. PUS defines 12 of these data types (identified by the Packet Field Type Code (PTC)), of which there are multiple variations (identified by the Packet Field Format Code (PFC)). Different PFC per packet field type offer options such as different integer bit-lengths, floating-point resolutions, or string lengths. We fully implemented all packet field type encodings using test-driven development, as the PUS documents offer detailed and testable serialization definitions.

In our library, we developed a serialization framework which includes an abstract *Serializer* class. Each packet field type implements a serializer class, which can encode a value into bytes, or decode bytes into a value. As PUS defines packet field types that serialize into odd numbers of bits, we incorporated mechanisms to support variable length bit-lengths and bit-offsets, which are always passed and processed together with serialized data. The data bits can then be correctly extracted from a byte string, and concatenated with other data.

In order to concatenate and extract values from concatenated data fields while keeping track of in-byte bit offsets, two helper classes were developed: the *Progressive Encoder* and the *Progressive Decoder*.

The *Progressive Encoder* provides functionality to sequentially construct a byte string composed of serialized data segments. It is initialized with an empty byte string, to which it subsequently appends data. To append a value, the value to be serialized and an appropriate serializer instance must be provided. The encoder then uses the given serializer to serialize the value and appends the resulting data to its internal byte string, correctly handling in-byte bit offsets during the process. It is used by services in order to built TM packet data, following a PUS-defined message structure with the associated packet field type serializers.

The *Progressive Decoder* operates in a similar manner, with the key difference that it is initialized with an existing byte string to be decoded. From this string, values can then be sequentially decoded using appropriate serializer instances. It is used by services to decode TC packet data according to PUS-defined message structures.

We considered an alternative approach during development, to define message structure classes which could be fully serialized and deserialized from and to python objects. However, the dynamic nature of certain message definitions made this approach overly complex. Consequently, we reverted to providing tools for sequential encoding and decoding, to be used within the service implementations.

PUS Components

The components within our PUS library serve as independent functional units which interface with the rest of the system through the interaction layer, thereby providing crucial functionality used by various services. Currently, these component instances are created and hosted by the application process, but – as they are accessed through component interfaces – they could be moved to independent processes and access by the PUS-Services could be replaced by proxy objects from the interaction layer.

Our library currently includes four such components:

- **Parameter Pool:** Collects and caches parameter values broadcasted over the interaction layer. As these values are of attributes of components interfaces, their names have to then be mapped to a corresponding parameter ID numbers, which is how parameters are addressed in PUS. This mapping is stored in the spacecraft database, which the parameter pool accesses.
- **Event Manager:** Similar to the parameter pool, the event manager monitors events broadcasted on the interaction layer. It looks up the event ID number and the event's severity classification as defined in the spacecraft database, and in turn broadcasts the event in severity channels with the correct ID for PUS service to listen to.
- **OBCP Engine:** Executes Onboard Control Procedure (OBCP) scripts and provides them with system access through the interaction layer. As we use PLUTO as our OBCP language, we implemented an engine that parses PLUTO scripts into Python scripts, and then executes them step by step. Apart from the correct execution of the scripts data and control flow, the engine must also provide access to system parameters, events and operations, which the scripts depend on. LibreCube had already developed such an engine, the *python-pluto-engine* [Lib24b], which however only implemented system interactions through HTTP calls. To integrate this engine with our interaction layer, we modified the engine to abstract system access to a *System Interface* class, with which we then implemented an adapter to access attributes, events and operations on the interaction layer. This was done using the *Transport* module of our interaction layer directly without proxy objects. For example a parameter access written in PLUTO "`angle of SolarPanel`" would be translated into a request on the interaction layer by calling the component function "`SolarPanel.angle.get`".
- **File Copy Service:** This component handles file copy and file transfer operations. It determines the type of copy operation (local, remote-to-local or local-to-remote) and interfaces with the CFDP protocol entity to coordinate file transfer operations. Further details on the file transfer implementation are provided in Section 3.1.3.

Services

The services implement the actual M&C functionality as defined in the PUS specification. They receive TC packets, process the contained instructions, and send data back in form of TM packets. During processing, a service generates verification reports, which provide feedback towards the ground segment over the progress and potential failures of a request. A skeleton code of such a service provider implementation can be found in Appendix B.

A notable special case is Service 1 – *Request Verification*, which routes TC packets to the destined service and monitors its execution through verification reports generated by the executing services. To route a TC packet, it looks up the service instance in the service lookup table by the service ID in the PUS packet header. It then calls its `handle_command` function in a new thread, so that command processing can be performed concurrently. This can be done in a safe manner, as any system interactions are performed over the interaction layer, which queues requests to be processed sequentially. If a service's command handler raises an exception during processing, it is catched by Service 1, which then generates a failure report TM packet to inform the ground segment about the failed execution. As Python does not offer a similar method to raise notifications about successful progress to a function caller, the Service 1 instead offers functions to generate success reports TM packets for service handlers to call. PUS defines four such steps in the processing of a command, for which such verification reports should be generated to inform about their success or failure:

1. *Acceptance:* The command was accepted and decoded by the corresponding service.
2. *Start:* The command parameters were verified and the command execution started.
3. *Progress:* Marks service-defined points of progress during processing; rarely used by PUS standard services.
4. *Completion:* The command processing was completed.

Some services require the persistent storage of context information, which should survive a system restart. For example, the event reporting service should store if the reporting for a given event is enabled or disabled. To this end, we implemented the functionality for services to safely load and store context data. As a sudden system shutdown could be triggered at any moment, the context storage operations must be protected from data corruption in this situation. We implemented this by leveraging atomic nature of the file rename operation of the Linux system [Lin24]. The context data is first written into a temporary file, with is then renamed to replace the context data file in an atomic manner.

We implemented the following services in our library, with more still under development:

- **Service 1 – Request Verification:** Handles command routing and the generation of verification reports as described previously.

- **Service 3 – Housekeeping:** Generates parameter reports at regular intervals. The report composition and reporting intervals can be configured through dedicated commands. It uses the parameter pool to collect current parameter values, and the cyclic loop function to autonomously trigger the report generation.
- **Service 5 – Event Generation:** Generates reports on onboard events with the option to enable and disable the report on specific events. It subscribes to the events channels of the event manager to get event reports classified with event IDs and severity.
- **Service 18 – OBCP:** Controls the execution of Onboard Control Procedures (OBCP) in the OBCP Engine, and allows for uploading of OBCP scripts.
- **Service 20 – Parameter Management:** Provides direct access to onboard parameters in order to read specific parameters, and to set parameters to new values. This service interacts with the parameter pool to execute these tasks.
- **Service 23 – File Management:** Executes operations on the onboard file system, and coordinates file transfers. The filesystem operations such as renaming, deletion, discovery and metadata analysis are implemented using the *os* module of the Python language, which in turn calls the corresponding Linux filesystem commands. For the file copy and transfer operations, the service commands the file copy service component, which then either performs a local file copy operation, or initiates a file transfer with CFDP.

To generate PUS telecommand packets as part of a test ground segment installation, we developed a *Service User* for each service. These service user instances provide functions to remotely call PUS service providers by sending TC packets, and they extract data from received TM packets and make them observable for test scripts. These service user instances can be created and registered in an application process in a similar fashion to service provider instances. The application process implementation supports the decoding and routing of TM packets to the designated service user instances, which is only used in the ground segment process.

3.1.3. File System and File Transfer with CFDP

In accordance with REQ 12 – *File Transmission with CFDP*, we chose the CCSDS File Delivery Protocol (CFDP) for file transmission between the onboard file system and the ground segment. In our system, its intended use is the transmission of files containing scientific data, OBCP code, or potentially system files in order to patch the OBSW in flight. LibreCube had already developed a Python library *python-cfdp* which implements the CFDP protocol with Space Packets transmitted through a UDP socket [Lib24a]. This implementation could be used without modification in our software stack by assigning a unique APID to the CFDP process and routing Space Packets with this APID to the CFDP port. However, as we intended to use PUS Service 23 *File Management* to initiate file transmissions, we had to develop an interface for the PUS service to communicate with the CFDP entity.

In terms of modularity and to enable parallel processing, CFDP should remain in a separate process from the PUS process. Consequently, communication between the two must occur via the interaction layer. For this reason, we developed a generalized component interface for remote file copy services, with an interface provider implementation for the CFDP entity. Through this component interface, the PUS service can command CFDP file transfers from and to ground, suspend, resume and abort operations, and get status information. Local file operations are handled by the PUS service itself and do not inquire the CFDP entity.

Strangely, the specification of PUS Service 23 states that the service can be used to coordinate file transfers between ground and spacecraft, but it does not provide a detailed definition of the corresponding commands intended for this purpose [Eur16]. Therefore, we considered how the existing commands of the service could be utilized to implement this functionality. The service defines a command to initiate a file copy operation, which requires the source file path and the target file path. We decided to extend the interpretation of these paths by introducing an optional prefix which differentiates between the file systems of the spacecraft or the ground segment. For example, to initiate a file transfer from space to ground, the MCS would send file copy command with the source file path `SAT:science/file.txt` and the target file path `GROUND:science/file.txt`. The spacecraft database includes a mapping from prefix to CFDP entity ID, which the PUS service then uses to determine the type of copy operation and – in case of a file transfer – passes the corresponding remote ID to the CFDP entity.

To abstract the folder structure of a Linux system and provide unified access to the onboard file system across the entire system, we aimed to implement a form of virtual file system. This virtual file system would serve as a dedicated location for files created and used by the OBSW during operation. Through our research, we identified a straightforward solution to this requirement: creating a dedicated directory within the system's file system and defining a system-wide environment variable that holds the path to this directory. Any application requiring access to this location can then read the environment variable and append the relevant file paths as needed. This approach enables the use of standard Linux file system operations and supports compliance with REQ 8 – *Use of Universal Features*.

3.1.4. Onboard Bus and Device Access

In our software stack, equipment access is handled by dedicated *Equipment Components* to leverage the flexibility provided by our interaction layer. These equipment components expose the equipment's functionality through attributes, operations and events in their component interface. The inner workings and communication handling of data acquisition and command execution over the underlying protocols with the onboard device is thereby abstracted away and simplified. For onboard software to monitor and command a device, it can simply call the interface operations and subscribe to attribute updates and events, without the need of implementation specific coding. This approach allows for rapid integration of new or

modified hardware modules into the system, which is a concern defined in CON 3 – *Testing Platform*. It also facilitates the reusability and exchangeability of hardware integrations, further satisfying REQ 4 – *Modularity*.

Equipment components could generally implement any means of device communication. However, as our system uses SpaceCAN as the main onboard bus, we implemented a *SpaceCAN Master* component which handles SpaceCAN communication and thereby simplifies the implementation of our equipment components. The SpaceCAN master handles encoding and decoding of SpaceCAN interactions over the underlying CAN bus with the Linux CAN driver *SocketCAN*.

The master exposes operations to generate SpaceCAN commands targeted at specific nodes and interpret their responses with long data strings fragmented over multiple CAN frames, as specified by the SpaceCAN protocol [Sch+19]. This basic SpaceCAN protocol handling is implemented in the *python-spacecan* library, previously developed by LibreCube [Lib25b].

SpaceCAN implements an optional service layer similar to PUS, which we decided to use as a generalized way of communication with onboard equipment. The services currently supported by SpaceCAN are equivalent to the respective PUS services, namely Service 1 – *Request Verification*, Service 3 – *Housekeeping*, Service 8 – *Function Management*, Service 17 – *Test* and Service 20 – *Parameter Management*.

Services 3 & 20 are used to get regular parameter updates and to request specific parameters. Service 8 is used to perform operations; this service is regarded as mostly obsolete in PUS as integrators favor custom services to perform custom operations as they offer grater flexibility. However, in SpaceCAN it makes sense to use this service in the context of specific device access, as device interactions are generally less complex so that the more limited nature of Service 8 functions is sufficient and offers a more unified interface.

Apart from generic functions to generate and receive SpaceCAN packets, the SpaceCAN master exposes functions to generate service-specific commands, and emits service specific response data as events. For example the master exposes the following function to request node parameters:

```python
def request_parameters(self, node_id: int, parameter_ids: List[int]):
    req_data = bytearray([len(parameter_ids)])
    req_data.extend(parameter_ids)
    self.send_command(node_id, 20, 1, req_data.decode("latin-1"))
```

This function builds the command's request data field, consisting of a length byte, followed requested parameter identifiers. The generation of the CAN frames to transmit the command with the SpaceCAN protocol is then handled by the more generic `send_command` function. Note the need to pass the `req_data`-bytes decoded into a character string using the bijective "latin-1" encoding, as this encoding supports the the full byte range. This is a result of the limitations of the JSON-based serialization approach we use in the interaction layer (see Section 3.1.1), which prevents the use byte-type arguments in functions exposed to the interaction layer.

Any command request performed by the SpaceCAN master automatically waits for a verification report returned by the responder node to confirm successful execution. In the event of a timeout or a failed verification report, the master raises an exception via the interaction layer's built-in error propagation mechanism. This exception, which includes the type of error and additional error data, is propagated to the calling process to enable appropriate handling of the failure.

Using the SpaceCAN master, the implementation of an equipment components is greatly simplified. The equipment component subscribes to the parameter report event of the SpaceCAN master and publishes any contained parameter values associated with the equipment as a component interface attribute update. When a equipment component's operation is called, it can simply call the master's `call_function` operation to execute the associated Service 8 function of the node. Potentially, equipment components could even be tool-generated using a mapping of SpaceCAN parameter IDs to component interface attributes, and SpaceCAN Service 8 functions to component interface operations.

3.2. Reconfiguration Module

As mentioned in Section 2.2, we decided to develop an optional hardware module that implements additional features of the SAVOIR architecture as part of this work. This *Reconfiguration Module* hosts the reconfiguration function, essential TC and TM, Safeguard Memory (SGM) and time reference. For our integrated system test, it also served as a hardware platform to validate the SpaceCAN communication with the OBC.

Requirements

The requirements for the newly developed module are as follows. The module must perform its processing using two hot-redundant controllers to ensure reliability. It shall provide interfaces to four UART buses - specifically, for each controller one connection to the Communications (COM) module and one to the Onboard Computer (OBC) module. Furthermore, the module must interface with the SpaceCAN bus, twelve system-control lines, a 5 V supply bus, and an arbitrary number of General Purpose Input / Output (GPIO) lines. The system-control lines are equivalent to the CPDU pulse lines of the SAVOIR FRA, as they enable or disable onboard hardware for reconfiguration.

For timekeeping, the module is required to host two precise clocks operating in hot redundancy, serving as the system's time reference. It shall also incorporate two EDAC-protected memory units in hot redundancy to host the SGM. Additionally, a power control switch unit must be included, capable of driving system-control lines and accessible by both controllers without collision.

Finally, the module must be capable of detecting processor failures and initiating

automatic recovery actions by restarting the affected processor to restore its functionality.

Parts Selection

We began the module's design process by identifying suitable Integrated Circuits (ICs) to meet the defined requirements. Although this design is intended for testing purposes only, we selected hardware that would viable to be used in a CubeSat system, in order to assess their suitability for the final system.

For the processing tasks, microcontrollers are required that supports two CAN interfaces and two UART interfaces. Additionally, the microcontroller should be easy to program for testing purposes and ideally compatible with a variety of development toolchains. The *Pyboard* was quickly identified as a strong candidate, as it is already used for testing within the LibreCube initiative, and several software modules have been developed specifically for it. The Pyboard v1.1 supports programming in MicroPython, a Python implementation designed for embedded systems. This also enables interactive debugging via a serial USB interface, which further facilitates development and testing. The board is a small development module based on the STM32F405RG microcontroller, that comes preloaded with a bootloader to run MicroPython code, but also supports other toolchains such as C/C++ with the STM32 HAL or the Arduino framework, or Rust with open-source frameworks such as the Embassy project [Emb].

For the Safeguard Memory (SGM), we chose the M24512-DRMN EEPROM, as it features a built-in Error Correction Code (ECC) and a relatively large capacity of 512 Kbits. This ECC can detect and correct flipped bits in its memory, which is automatically applied on read operations. However, it only supports raw, unprotected data transmission via I^2C, which means that data corruption during transmission remains a possibility. Therefore, additional error correction mechanisms should be considered, such as appending Hamming codes to data chunks to ensure integrity.

For the reference time source, the DS3231SN real-time clock was selected. This device integrates a Temperature-Compensated Crystal Oscillator (TCXO), thereby reducing design complexity by eliminating the need for an external oscillator. Additionally, it provides programmable alarm functions that can assert interrupt signals, enabling accurate time synchronization. The DS3231SN also supports an optional backup power supply input, ensuring reliable timekeeping in the event of a main power failure. This backup supply can be provided either by a dedicated battery or by a redundant power rail from the Power Control and Distribution Unit (PCDU). An other option we considered was to use an elapsed time counter instead of a real-time clock, as the transmission of a timestamp is more complex than that of a counter value. However, due to the limited options of accurate elapsed time counter ICs, we decided to use a clock for this test revision module instead.

To detect processor failures, we decided that a simple watchdog timer implementation would be sufficient for our cause. A watchdog timer listens for heartbeat

signals that have to be sent in regular intervals by the processor, triggered in the main loop of the software. If the software hangs or the processor enters an error state, the heartbeat signal is not sent and the watchdog timer resets the processor by pulling down its reset line. We selected the TPS3813K33 processor supervisory circuit with window-watchdog for this task, as it offers great customizability regarding the heartbeat window timings.

For the power control switch unit, we had to design a custom circuit which lets us control multiple output lines independently by two controllers. This unit resembles the Command Pulse Decoding Unit (CPDU) of the SAVOIR architecture, with the major difference that the CPDU specifically distributes electrical pulses, whereas our unit is intended to distribute steady signals which control the on / off state of the onboard equipment. In lack of a better name, we regardless adopted the name "CPDU" for our power control switch unit in some parts of this work.

SAVOIR proposes to keep the complexity of the CPDU to a minimum in order to ensure maximum reliability, so we decided to implement it using simple flip-flops. The processors take turns at writing to the flip-flops, whose outputs subsequently drive the system-control lines. In order to keep the number of interfacing lines between flip-flops and processors to a minimum, we decided to use D-type flip-flops, as they only require one input line per gate, and a data transfer can be triggered by a shared clock line. For ease of debugging and manufacturing, we decided to use two CD74HCT273E octal D-type flip-flops in the large DIP housing. As by the design requirements, we only use 12 of the resulting 16 flip-flop gates. To decouple the potential load on the system-control lines from the flip-flop outputs, we decided to use three CD4066BE quad bilateral switches.

Circuits

The Pyboard controller operates at 3.3 V, while the available power rail of the LibreCube platform provides 5 V. To accommodate this, the Pyboard is equipped with an onboard Low-Dropout (LDO) regulator that converts the 5 V supply to 3.3 V. Since the Pyboard's I/O signal levels are at 3.3 V, all circuitry on the module was designed to operate at the same voltage level. Consequently, a reliable 3.3 V power source is required. Given that both Pyboards independently generate their own 3.3 V supply, we opted to combine them into a single 3.3 V rail powered by both controllers. To achieve this, two MAX40200AUK ideal diodes were used to safely merge the outputs without introducing backfeeding or contention between the two sources.

As SpaceCAN requires two Controller Area Network (CAN) connections per node, we added two transceivers to each controller that translate the differential signal to CMOS-level signals. Additionally, each controller was connected to one EEPROM and one clock via an I^2C-bus.

For the CPDU, we connected 12 GPIO lines from the controllers to the flip-flops D-input pins, and one line to both clock pins. There is a risk of both controllers trying to access the CPDU at the same time, which would lead to an inconsistent state and

short-circuits on mismatching signals. For this reason, we implemented two safety features to avoid signal collisions:

1. To avoid short-circuits on conflicting signal levels, we connected the nominal controller directly to the flip-flops, and the redundant controller through resistors. If they set their outputs at the same time to conflicting states, only a small current flows through the resistors, and the signals of the nominal controller have priority.
2. To avoid an access conflict in the first place, we implemented Dekker's algorithm [Dij65] to ensure mutually exclusive access.

In Dekker's algorithm, both processors raise flags when they want to access a shared resource. In our implementation, those flags are represented by request signal lines (`CPDU_REQ`). SAVOIR states, that on a conflict of CPDU access, the controllers have to negotiate based on a determined command priority [Eur21]. As Dekker's algorithm follows turn-taking instead of priority based negotiation, we had to modify it to fit our needs. To get access to the CPDU, our algorithm follows these steps:

1. Wait until the flag of the other controller is lowered.
2. Raise the own flag and wait until capacitive loads are settled.
3. If the other controller has not risen its flag, access is now granted.
4. Otherwise, enter a battle for priority: Wait an amount of time, dependant on the priority.
5. Check if the other flag is still up. If it is, the battle is lost, the other controller is granted access and this controller has to retry. If the other flag is down, the battle is won and access is granted.

To avoid both controllers entering a battle with equal priority, the nominal controller always has a slightly higher base priority than the redundant one.

One hazard of Dekker's algorithm in our application would be, that if a controller due to an error never lowers its request signal, the other controller also waits indefinitely and can't access the CPDU either. As the module could not recover from this condition, it represents a single point of failure that we want to avoid. For this reason we implemented an additional safety circuit, which detects if a request signal is stuck on high, and in that case resets the controller. The request signal charges a capacitor through a resistor, with the capacitor connected to a Schmitt trigger input. If the request signal remains high for an extended period, the capacitor charges up to the high threshold voltage of the Schmitt trigger. Once this threshold is reached, the Schmitt trigger drives the controller's reset line low via an N-channel MOSFET, initiating a reset. During the reset process, the controller places its request pin in a high-impedance state, allowing the capacitor to discharge through an additional resistor. When the voltage across the capacitor drops below the Schmitt trigger's lower threshold, the reset signal is released, and the controller enters its boot sequence.

The complete schematic of the module can be found in Appendix C.

Evaluation

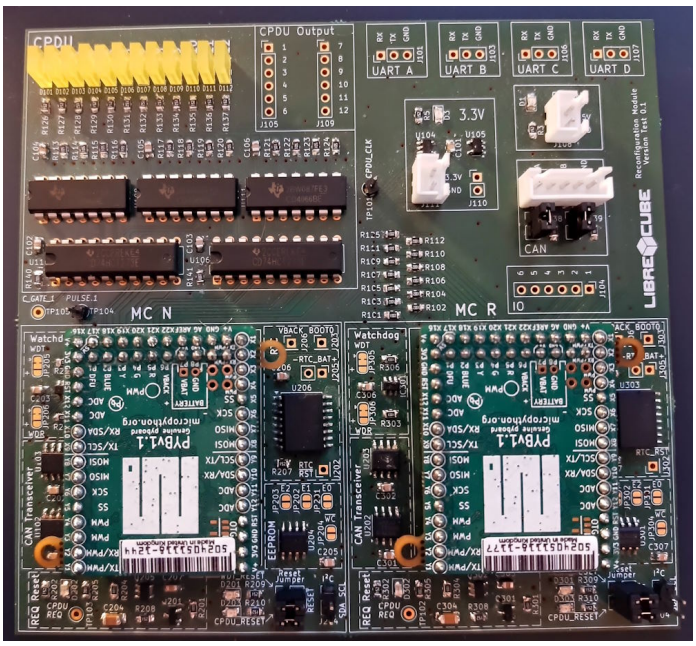


Figure 3.5.: *Reconfiguration Module Circuit Board*

To evaluate our circuit design, we designed a PCB using KiCad (see Figure 3.5), which was then manufactured and tested for functionality. All circuits performed as expected, and we encountered only two problems with our electrical design:

1. The flip-flops encountered spurious gate flips when no controller was actively driving the gate inputs and clock signal. The main cause of this was the lack of a pull-down resistor on the clock line, which leads to random floating gate behavior. But even when a pull-down resistor was added, the problem persisted. Only after the addition of a ceramic capacitor on the clock line to absorb low-energy voltage spikes, the problem was mitigated. The root cause of these spikes is left undetermined, but the floating gate input lines may contribute to this unexpected behavior, which is why the design should be improved by additional pull-down resistors on the input lines.
2. As the control pulse lines of both controllers are coupled through 10 kΩ resistors, the second controller has to drive the flip-flop gates through these resistors. Due to the increased pin capacitance of the large DIP package flip-flops, the signal rise timings as specified in the datasheet were not sufficient and had to be increased significantly. This problem could be improved by choosing smaller coupling resistors, different coupling approaches, and smaller flip-flop packages.

In order to test the SpaceCAN communication, we installed LibreCube's MicroPython library *micropython-spacecan* on the Pyboards [Lib25a], which we then configured to act as SpaceCAN responders. With this approach, we could successfully establish SpaceCAN communication with our Raspberry Pi OBC.

4. Experimental Evaluation

4.1. System Integration Test

In order to test all capabilities of our system in practice, we built a small integrated setup, consisting of an Onboard Computer (OBC) and our reconfiguration module acting as a responder node (see Figure 4.1).

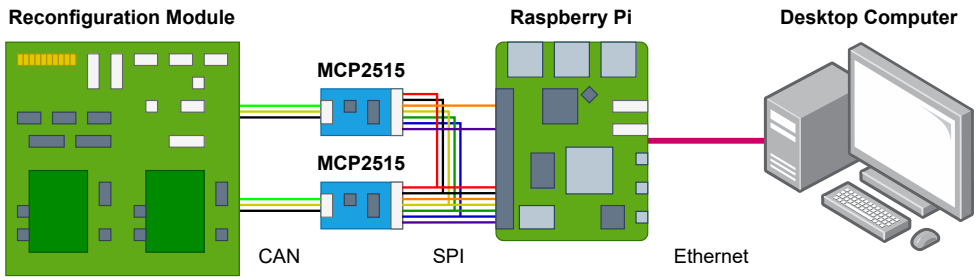


Figure 4.1.: *System Integration Test Layout*

For the OBC, we used a Raspberry Pi 5 computer, running the Raspberry Pi OS Lite with our software stack installed. The Lite variant includes only a minimal set of pre-installed software and includes no desktop environment, which is however not needed for our purpose. Instead, we use SSH connections and the remote development capabilities of the Visual Studio Code editor to deploy and monitor our software.

The OBC and reconfiguration module communicate via SpaceCAN, which requires two CAN connections. As the Raspberry Pi does not feature built-in CAN capabilities, we used two MCP2515 modules, which translate the CAN bus to SPI.

Both MCP2515 modules are connected to the SPI pins on the Raspberry Pi's GPIO header with different Chip Select (CS) pins. The Raspbian operating system features a built-in driver for the MCP2515 module, which has to be loaded by a corresponding modification to the boot configuration file. After that, the can interfaces are available as network devices in the Linux system, which can then be set-up as network sockets with the `ip link` command.

Our software stack requires Python of version 3.10 or newer, which is installed by default in the Raspbian operating system.

On the software side, we developed a small *System Test* package which includes configurations and components to operate our system set-up with our onboard software

stack. It serves as an example on how to set up our software on a real avionics system. This test however is a bit simplified compared to how an installation on a real integrated spacecraft system is planned: we are feeding Space Packets directly to the application processes instead of the planned CCSDS Frame communication over UART. This was done as the frame processor component – responsible for the unpacking of frames – is still under development during time of writing.

The system test package consists of the following files:

- **`sdb.py`:** This file poses as the spacecraft database. It includes component interface classes, parameter and event mappings, the APIDs of the space and ground segment processes, and the ZeroMQ ports of all components.
- **`cpdu.py`:** In this file, the equipment component interface provider of the reconfiguration module is implemented. For this test set-up, we only implemented the functionality to control and monitor the state of CPDU lines.
- **`controller.json`:** This file is used by the *python-spacecan* library to initialize the SpaceCAN master controller. It includes the definition of the CAN network sockets to use, and the configuration of the heartbeat generation.
- **`services_cpdu.json`:** This file defines the parameters, functions, and housekeeping report structures provided by the CPDU responder node. The SpaceCAN master uses this file to look up the data type of parameters and function arguments in order to choose the correct serialization.
- **`test_sys.py`:** This script serves as the entry point for the software, as it configures and initiates all components of the software stack.
  First, it initializes the CFDP entity along with its UDP socket. Subsequently, an instance of the interaction layer is created to facilitate communication between the component instances. In the future, the platform management component will launch each component in a separate process, necessitating individual interaction layer instances. However, for this initial test setup, a single process – and thus a single interaction layer instance – is sufficient.
  Once the interaction layer is initialized, the script starts the SpaceCAN master and the equipment components for both the nominal and the redundant CPDU.
  Following this, the PUS module is set up. This involves initializing the Space Packet protocol entity and its associated UDP socket. These, together with the spacecraft database, the interaction layer instance, and the APID, are passed to the constructor of the PUS application process. The relevant service provider instances are then instantiated and registered with the application process.
  At this stage, all components are fully initialized, and the script proceeds to execute the main loop of the PUS application process, which runs continuously until the program is terminated.

We tested this setup by running a ground segment PUS process on a separate computer, which connected to the onboard PUS process through UDP. The ground segment runs a script, which runs the service user functions to test the various PUS services. The most advanced test of this routine is to start an OBCP from ground, which then calls the CPDU to set its lane states. This test demonstrates how al-

most all parts of our software work together as the commanding has to successfully traverses through all communication layers:

1. A TC Space Packet is generated by the PUS ground process to command the execution of an OBCP script.
2. The TC packet is sent to the onboard PUS process through UDP.
3. The TC packet is decoded and interpreted by the onboard application process and the OBCP service.
4. The OBCP service commands the OBCP engine component to start the procedure.
5. The OBCP engine executes the PLUTO script, which contains an instruction to "`initiate and confirm set_state_bits of CpduN`".
6. This activity call is converted by the *System Interface* adapter to an interaction layer remote function call of "`CpduN.set_state_bits`".
7. The interaction layer instance then sends a ZeroMQ request message to the response port of the CpduN equipment component.
8. The CpduN equipment component then calls the `call_function` operation of the SpaceCAN master component.
9. The SpaceCAN master then generates the corresponding CAN messages to the CpduN responder node on the Pyboard of the reconfiguration module.
10. The CpduN responder node implementation receives the messages and then calls the user-defined `set_state_bits` function.
11. In this function, the Pyboard then acquires CPDU access according to our implementation of the Dekker's algorithm, sets its GPIO pins according to the commanded state, and toggles the flip-flop clock line.
12. The flip-flops adopt the commanded gate inputs and latch their outputs accordingly, so that the commanded state can be observed on the output LEDs.

While this communication chain may appear extensive, it is simply a consequence of the abstraction layers required to achieve the targeted level of modularity as defined by the relevant standards. Introducing shortcuts in this chain would necessitate deeper implementation-specific knowledge and implicit assumptions between system components, leading to tighter coupling, reduced modularity, and increased overall complexity. Conversely, when these communication mechanisms operate reliably, the minimal interfaces at the application layer help to reduce system complexity and enhance extensibility.

In addition to the OBCP test, other significant test cases include successful file transfers via CFDP as well as parameter and event reporting. All tests have consistently performed reliably within our test environment, demonstrating the effective integration of the system components.

4.2. Compatibility with Mission Control Software

To verify that our system is capable of communicating with dedicated mission control systems, we connected it to an instance of the open-source mission control system *Yamcs* [Spa]. Yamcs supports a variety of communication protocols and includes functionality for handling Space Packets, and specifically the decoding of PUS packets. It can establish connections via network sockets, such as User Datagram Protocol (UDP).

Since our onboard software transmits and receives packets via UDP ports, connecting Yamcs to our system was straightforward. Out of the box, we were able to receive and display PUS packets generated by our software. This confirms that the relevant protocols are correctly implemented and processed by our system and that it would be compatible with European satellite control terminals.

To configure Yamcs for displaying mission-specific data such as parameter values or sending mission-specific commands, it utilizes configuration files in the *XTCE* format. XTCE stands for XML Telemetric and Command Exchange; it is a standardized configuration file format that defines the structure of telemetry and telecommand data, as specified by the CCSDS [Con21c]. Since its structure closely resembles the *spacecraft database* used in our system, the development of an automated conversion tool could be considered for future work.

5. Conclusions

5.1. Summary

In this work, we successfully developed a functional implementation of a spacecraft onboard software stack. Our open-source implementation demonstrates the feasibility of employing European and international architecture standards and communication protocols in CubeSat applications. We adapted the relevant specifications to better suit the simplified requirements typical of CubeSat systems and introduced a novel approach to onboard device communication, further streamlining system integration.

The software stack offers a modular design that facilitates rapid prototyping, as the integration and modification of hardware components is greatly simplified through the use of generalized and minimal component interfaces. It will be adopted by LibreCube to support the development of their CubeSat hardware platform, with an integration into the LibreCubeRover planned in the near future.

Moreover, our software stack serves as a demonstration platform for CubeSat developers and students, enabling them to explore the structure of modular onboard software systems. In the long term, the architecture is intended to serve as a reference for a robust, space-capable embedded implementation – written in Rust and/or C++ – designed to run on a real-time operating system.

Our software represents the first open-source implementation based on the European SAVOIR Onboard Software Reference Architecture (OSRA), integrating both the monitoring and control system PUS and the recently published file transfer protocol CFDP. We also proposed a novel hardware module that can be integrated into typical CubeSat hardware platforms to incorporate reliability features of the SAVOIR Functional Reference Architecture (FRA). By incorporating these agency-level standards, CubeSat systems can benefit from the proven, reliable foundation these standards offer, serving as a basis for developing robust and dependable systems. This integration brings the CubeSat developer community and institutional space agencies closer together, facilitating more effective knowledge transfer across these traditionally separate domains.

The newly developed SpaceCAN onboard bus system provides a reliable and cost-effective alternative to the traditionally unreliable onboard communication systems used in CubeSat platforms. Our software stack demonstrates that this bus system integrates seamlessly into onboard software architectures and even supports tool-generated equipment integrations with only minimal configuration effort. This ap-

proach promotes the reusability and compatibility of hardware modules developed by different suppliers – an aspect that has traditionally posed significant challenges for CubeSat developers.

5.2. Limitations

While our software stack serves as a functional implementation, it should not be considered viable for use in space. Beyond the fact that the system components have not undergone the rigorous testing required for deployment on an actual satellite, the use of Python in space systems presents critical drawbacks. As an interpreted language, Python imposes a significant overhead in system resource usage compared to implementations in compiled languages such as C++ or Rust: studies have shown that Python is, on average, three to four times slower than C++ [Pre00], [Zha24]. Given the limited energy budgets available on CubeSats, the resulting increase in energy consumption by the OBC could present a serious constraint.

However, our software is already usable for deployment on prototype systems. To fully integrate communication with the antenna system based on CCSDS frames, the development of the frame processor must be completed. Certain implementation details currently impose minor constraints on the component interface definitions, such as the use of JSON serialization and the rigid port assignments in our implementation of the interaction layer. It remains to be seen whether these constraints will pose actual challenges during prototype development; if so, appropriate refinements can be made.

Due to the modular nature of our software stack, any limiting components can be replaced with alternative implementations to overcome existing constraints. Furthermore, it would even be feasible to gradually substitute all components with implementations in compiled languages, thereby transforming the system into a software stack suitable for deployment in a real satellite system.

5.3. Directions for Future Work

As previously discussed, the development of the proposed software stack will be continued in order to further enhance its capabilities.

Planned extensions include:

- Integration of CCSDS frame encoding and decoding within the frame processor component.
- Implementation of additional PUS services, such as Service 2 – *Device Access* and Service 15 – *Onboard Storage and Retrieval*.
- Development of application components to demonstrate their interaction with our execution platform.

- Design of a robust high-speed communication protocol over the payload UART bus for onboard transfer of scientific data.
- Full integration of the reconfiguration module and its functionalities into the software stack.

Additionally, the reconfiguration module will be further refined with the aim of integrating it into the LibreCube hardware platform as a CubeSat module, compliant with standard size and interface specifications.

Looking ahead, we anticipate porting our software stack to a space-ready implementation based on Rust and FreeRTOS, intended for deployment on a CubeSat utilizing the LibreCube hardware platform. This work thus serves as an architectural guideline, an implementation reference, and a functional comparison system for verification purposes.

List of Acronyms

List of Figures

A. PUS Packets and Services

The primary Space Packet header includes the following fields [Eur16]:

Packet Version Number	Number of the space packet protocol version in use.
Packet Type	Differentiates TC and TM packets.
Secondary Header Flag	States if the packet has a secondary header.
Application Process Identifier (APID)	Identifies the targeted process.
Sequence Flags	Used for sequencing, but unused by PUS.
Packet Sequence Count	Number that increments per packet, used to uniquely identify packets.
Packet Data Length	Number of bytes in the data field (including secondary header).

PUS defines a secondary header to be used on its message packets, which is different on TC and TM packets.

The PUS secondary header on TM packets includes the following fields [Eur16]:

PUS Version Number	Number of the PUS version in use.
Spacecraft Time Reference Status	Indicates the quality status of the onboard time, e.g., synchronization.
Service Type ID	ID of the service that generated this packet.
Message Subtype ID	ID of the message type of this packet.
Message Type Counter	Counter that increments per message sent of this type.
Destination ID	APID of the destination process to receive this packet.
Time	Onboard timestamp of the time this packet was generated.

The PUS secondary header on TC packets includes the following fields [Eur16]:

PUS Version Number	Number of the PUS version in use.
Acknowledge Flags	Indicates if acknowledge reports shall be generated throughout the processing of this request.
Service Type ID	ID of the service that generated this packet.
Message Subtype ID	ID of the message type of this packet.
Source ID	APID of the process that generated this packet.

PUS defines the following services [Eur16], [Eic12]:

1	*Request Verification:* Sends status reports on the progress and success of request processing.
2	*Device Access:* Provides direct onboard hardware access.
3	*Housekeeping:* Sends periodic parameter reports.
4	*Parameter Statistics Reporting:* Provides statistics on onboard parameters.
5	*Event Reporting:* Reports onboard events.
6	*Memory Management:* Provides direct access to onboard memories.
8	*Function Management:* Executes mission-defined functions.
9	*Time Management:* Onboard time reporting. Often extended by custom time synchronization.
11	*Time-Based Scheduling:* Schedules and executes lists of prepared commands at defined times.
12	*On-Board Monitoring:* Monitors onboard parameters and sends out-of-limit reports.
13	*Large Packet Transfer:* Transmission of large data.
14	*Real-Time forwarding Control:* Configures realtime transmission of reports on ground contact.
15	*Onboard Storage and Retrieval:* Configures transmission of stored reports.
17	*Test:* Performs a simple connection test.
18	*On-Board Control Procedure:* Upload and execution of OBCPs.
19	*Event-Action:* Configures automatic actions based on onboard events.
20	*Parameter Management:* Sets and redefines onboard parameters.
21	*Request Sequencing:* Upload and execute sequences of TC packets.
22	*Position-Based Scheduling:* Upload and execute commands and defined positions.
23	*File Management:* Copy, delete and discover onboard files.

B. PUS Service Skeleton Code

```python
class ServiceName(ServiceProvider):
    SERVICE_ID = 42 # put service ID here

    def __init__(
        self,
        process: "ApplicationProcess",
    ):
        super().__init__(process)

    def handle_command(self, packet: PusTcPacket) -> bool:
        """Call the appropriate handler function to handle the TC packet.
        Return True if package was handled."""

        msg_id = packet.message_subtype_id
        if msg_id == 1:
            # put TC handlers here
            self.tc_handler_name(packet)
        else:
            return False

        return True

    def tc_handler_name(self, packet: PusTcPacket):
        # Decode
        decoder = ProgressiveDecoder(packet.application_data_field)
        # do decoding here

        # raise AcceptError(
        #     packet, FailureCode.
        # )
        self.rv.send_success_acceptance_report(packet)


        # Validate
        # do validation here

        # raise StartError(
        #     packet, FailureCode.
        # )
        self.rv.send_success_start_report(packet)


        # Execute
        # execute the instructions here

        # raise CompletionError(
        #     packet, FailureCode.
        # )
        self.rv.send_success_completion_report(packet)
```

```python
class ServiceNameUser(ServiceUser):
    SERVICE_ID = 42 # put service ID here

    def __init__(
        self,
        process: "ApplicationProcess",
        provider_apid: int,
    ):
        super().__init__(process, provider_apid)

        self.example_observable = Observable[int]()


    def handle_telemetry(self, packet: PusTmPacket):
        """Call the appropriate handler function to handle the TM packet."""

        msg_id = packet.message_subtype_id
        if msg_id == 2:
            # put TM handlers here
            self.tm_some_message(packet)


    def some_command(
        self,
    ) -> Queue[VerificationReport]:
        # do some preparation work here

        # Encode
        encoder = ProgressiveEncoder()
        # encode the TC message here

        # send TC packet and create verification report receiver queue
        return self.start_transaction(42, 1, self.provider_apid, encoder.data)


    def tm_some_message(self, packet: PusTmPacket):
        # Decode
        decoder = ProgressiveDecoder(packet.source_data_field)
        # decode the message here

        # Publish
        # publish received data to observables
        self.example_observable.fire(42)
```

C. Reconfiguration Module Schematics

The following schematics of our reconfiguration module as described in Section 3.2 were created with KiCad 9.0.0. They utilize the hierarchical sheets feature of KiCad: since some hardware of this module is duplicated to provide both nominal and redundant instances, we created a *controller* sheet containing all redundant hardware. This sheet is then instantiated twice as `MC_N` and `MC_R` in the outermost Sheet 1. Interfacing connections into and out of the nested controller sheet are highlighted as labels with arrows on Sheet 2.

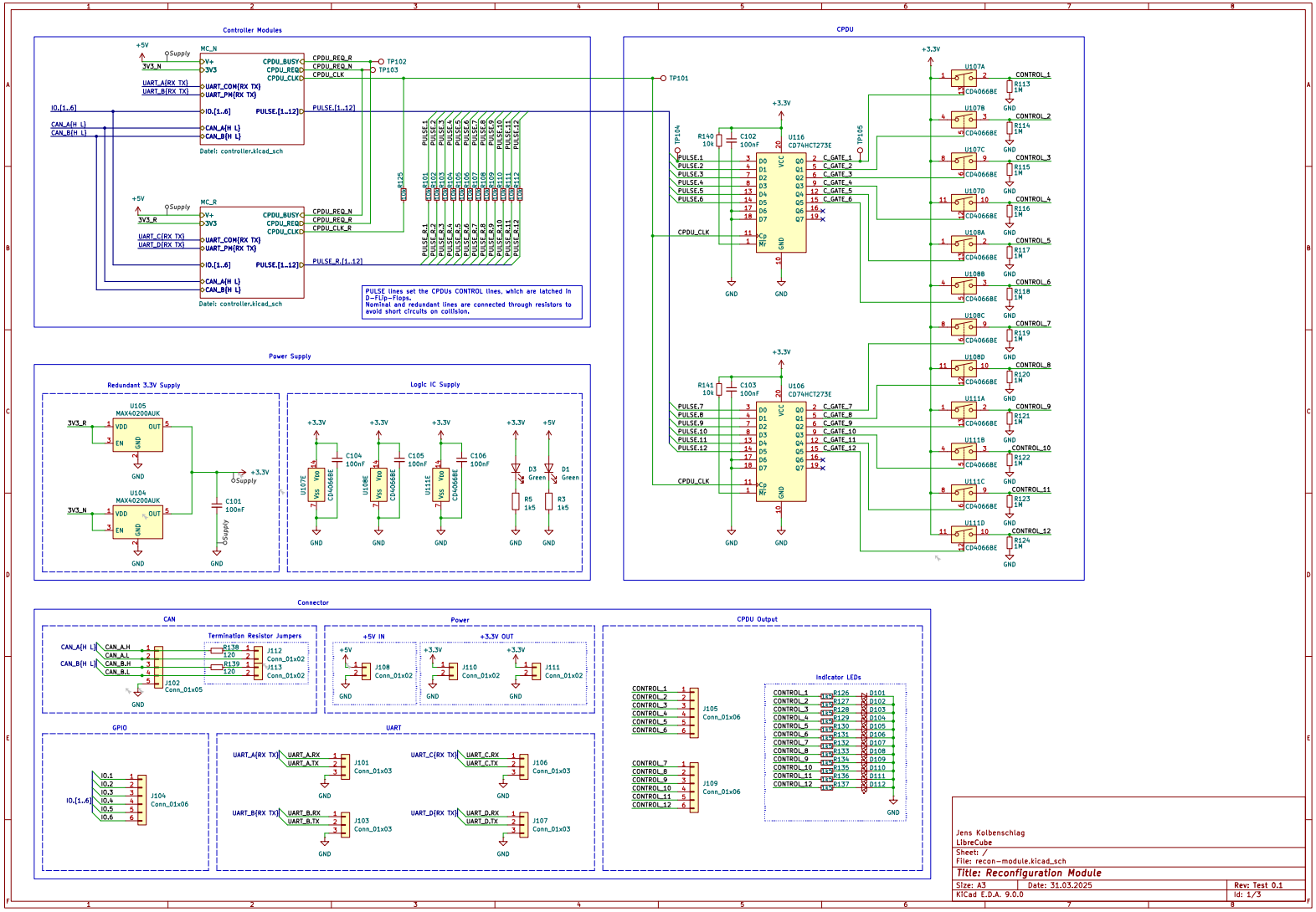


Figure C.1.: *Reconfiguration Module Schematics Sheet 1*

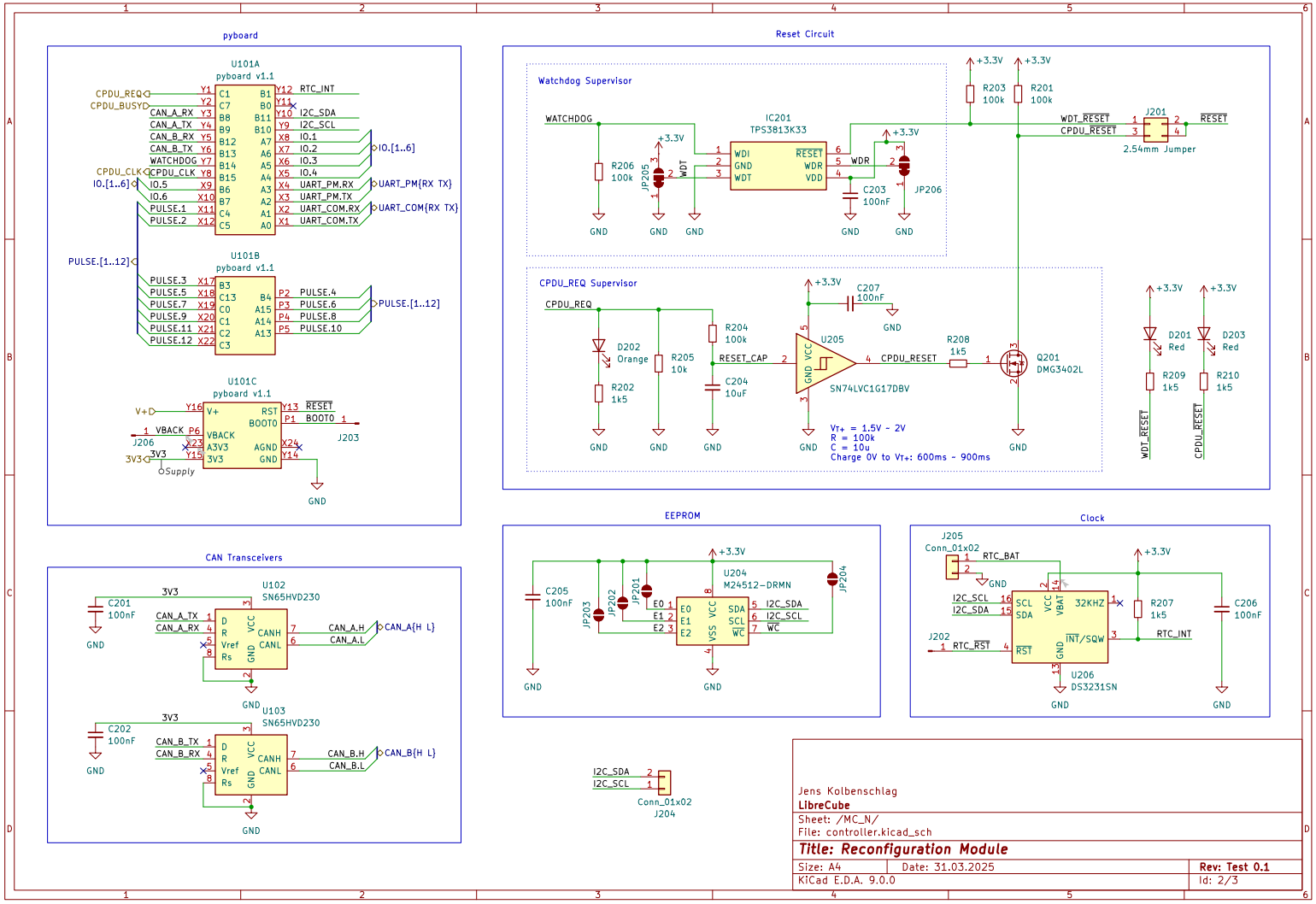


Figure C.2.: *Reconfiguration Module Schematics Sheet 2: Controller Module* `MC_N` *(redundant module* `MC_R` *similar)*

Bibliography

[AGW10] Jan Andersson, Jiri Gaisler, and Roland Weigand. "Next generation multipurpose microprocessor". In: *Int. Conf. on Data Systems in Aerospace (DASIA), Hungary*. 2010.

[Bes18] Valentin Le Bescond. *ZeroMQ vs DBus for Pub-Sub pattern*. Jan. 2018. URL: https://hackaday.io/project/279-sonomkr-noise-monitoring/log/86364-zeromq-vs-dbus-for-pub-sub-pattern (visited on 06/15/2025).

[BLG17] Jasper Bouwmeester, Martin Langer, and Eberhard Gill. "Survey on the implementation and reliability of CubeSat electrical bus interfaces". In: *CEAS Space Journal* 9.2 (2017), pp. 163–173. ISSN: 1868-2510. DOI: 10.1007/s12567-016-0138-0. URL: https://doi.org/10.1007/s12567-016-0138-0.

[Cho+20] Mengu Cho, Takashi Yamauchi, Marloun Sejera, Yukihisa Ohtani, Sangkyun Kim, and Hirokazu Masui. "CubeSat Electrical Interface Standardization for Faster Delivery and More Mission Success". In: *Proceedings of the 2020 AIAA/USU Conference on Small Satellites*. 2020. URL: https://digitalcommons.usu.edu/smallsat/2020/all2020/3/.

[Con13] Consultative Committee for Space Data Systems (CCSDS). *Spacecraft Onboard Interface Services. CCSDS 850.0-G-2, Green Book, Issue 2*. Dec. 2013. URL: https://ccsds.org/wp-content/uploads/gravity_forms/5-448e85c647331d9cbaf66c096458bdd5/2025/01//850x0g2.pdf (visited on 05/15/2025).

[Con20a] Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP). CCSDS 727.0-B-5, Blue Book, Issue 5*. July 2020. URL: https://public.ccsds.org/Pubs/727x0b5e1.pdf (visited on 05/15/2025).

[Con20b] Consultative Committee for Space Data Systems (CCSDS). *Space Packet Protocol. CCSDS 133.0-B-2, Blue Book, Issue 2*. Recommended Standard. June 2020. URL: https://public.ccsds.org/Pubs/133x0b2e2.pdf (visited on 05/15/2025).

[Con21a] Consultative Committee for Space Data Systems (CCSDS). *TC Space Data Link Protocol. CCSDS 232.0-B-4, Blue Book, Issue 4*. Oct. 2021. URL: https://ccsds.org/Pubs/232x0b4e1c1.pdf (visited on 06/14/2025).

[Con21b] Consultative Committee for Space Data Systems (CCSDS). *TM Space Data Link Protocol. CCSDS 132.0-B-3, Blue Book, Issue 3*. Oct. 2021. URL: https://ccsds.org/Pubs/132x0b3.pdf (visited on 06/14/2025).

[Con21c] Consultative Committee for Space Data Systems (CCSDS). *XML Telemetric and Command Exchange (XTCE). CCSDS 660.2-G-2, Green Book, Issue 2*. Tech. rep. Informational Report. Feb. 2021. URL: https://ccsds.org/Pubs/660x2g2.pdf (visited on 06/14/2025).

[Cub22] The CubeSat Program, Cal Poly SLO. *CubeSat Design Specification Rev. 14.1*. Tech. rep. Revision 14.1. California Polytechnic State University, Feb. 2022. URL: https://www.cubesat.org/s/CDS-REV14_1-2022-02-09.pdf (visited on 05/15/2025).

[Dij65] E. W. Dijkstra. "Solution of a Problem in Concurrent Programming Control". In: *Communications of the ACM* 8.9 (1965), pp. 569–573.

[Eic12] Jens Eickhoff. *Onboard Computers, Onboard Software and Satellite Operations*. Jan. 2012. ISBN: 978-3-642-25169-6. DOI: 10.1007/978-3-642-25170-2.

[Emb] Embassy project contributors. *Embassy Project*. URL: https://embassy.dev/ (visited on 05/30/2025).

[Eura] European Space Agency (ESA). *Architectures of Onboard Data Systems*. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Architectures_of_Onboard_Data_Systems (visited on 05/16/2025).

[Eurb] European Space Agency (ESA). *European Space Software Repository*. URL: https://essr.esa.int/ (visited on 05/19/2025).

[Eur08] European Cooperation for Space Standardization (ECSS). *ECSS-E-ST-70-32C: Space Engineering – Test and Operations Procedure Language*. Defines the PLUTO language for automated test and operations procedures. July 2008. URL: https://ecss.nl/standard/ecss-e-st-70-32c-test-and-operations-procedure-language/ (visited on 05/15/2025).

[Eur10] European Cooperation for Space Standardization (ECSS). *ECSS-E-ST-70-01C: Spacecraft on-board control procedures*. Standard. Apr. 16, 2010. URL: https://ecss.nl/standard/ecss-e-st-70-01c-on-board-control-procedures/.

[Eur15] European Cooperation for Space Standardization (ECSS). *ECSS-E-ST-50-15C: CANbus extension protocol*. Standard. May 1, 2015. URL: https://ecss.nl/standard/ecss-e-st-50-15c-space-engineering-canbus-extension-protocol-1-may-2015/.

[Eur16] European Cooperation for Space Standardization (ECSS). *ECSS-E-ST-70-41C: Space Engineering – Telemetry and Telecommand Packet Utilization*. Standard ECSS-E-ST-70-41C. Apr. 2016. URL: https://ecss.nl/standard/ecss-e-st-70-41c-space-engineering-telemetry-and-telecommand-packet-utilization-15-april-2016/.

[Eur21] European Space Agency (ESA). *Space Aionics Open Interface Architecture (SAVOIR)*. Issue 1 Rev 2. Nov. 2021. URL: https://savoir.estec.esa.int/ (visited on 05/15/2025).

[fre] freedesktop.org Project. *D-Bus Homepage*. URL: https://freedesktop.org/wiki/Software/dbus/ (visited on 06/15/2025).

[Hin] Pieter Hintjens. *ØMQ - The Guide*. URL: https://https://zguide.zeromq.org/ (visited on 06/01/2025).

[Jet] Jet Propulsion Laboratory. *F Prime*. URL: https://fprime.jpl.nasa.gov/ (visited on 06/15/2025).

[Jon21] Irmen de Jong. *PYRO4 - Python Remote Objects*. https://github.com/irmen/Pyro4. 2021.

[Kul24] Erik Kulu. "CubeSats & Nanosatellites - 2024 Statistics, Forecast and Reliability". In: *75th International Astronautical Congress*. Nanosats Database. IAF, Oct. 2024.

[Lib] LibreCube e.V. *LibreCube Docs Website*. URL: https://librecube.gitlab.io/ (visited on 05/15/2025).

[Lib24a] LibreCube e.V. *Python CFDP*. https://gitlab.com/librecube/lib/python-cfdp. 2024.

[Lib24b] LibreCube e.V. *Python Pluto Engine*. https://gitlab.com/librecube/prototypes/python-pluto-engine. 2024.

[Lib24c] LibreCube e.V. *Python Space Packet*. https://gitlab.com/librecube/lib/python-spacepacket. 2024.

[Lib25a] LibreCube e.V. *MicroPython SpaceCAN*. https://gitlab.com/librecube/lib/micropython-spacecan. 2025.

[Lib25b] LibreCube e.V. *Python SpaceCAN*. https://gitlab.com/librecube/lib/python-spacecan. 2025.

[Lik+10] Justin Likar, Stephen Stone, Robert Lombardi, and Kelly Long. "Novel Radiation Design Approach for CubeSat Based Missions". In: *Proceedings of the 2010 AIAA/USU Conference on Small Satellites*. 2010. URL: https://digitalcommons.usu.edu/smallsat/2010/all2010/15/.

[Lin24] The Linux Foundation. *rename(2) Linux User's Manual*. Version 6.10. July 2024.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[Nat] National Aeronautics and Space Administration (NASA). *core Flight System (cFS)*. URL: https://etd.gsfc.nasa.gov/capabilities/core-flight-system (visited on 06/15/2025).

[Org19] Organization for the Advancement of Structured Information Standards (OASIS). *MQTT Version 5.0*. Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (visited on 05/30/2025).

[PC108] PC/104 Consortium. *PC/104 Specification Version 2.6*. Tech. rep. Revision 2.6. PC/104 Consortium, Oct. 2008. URL: https://pc104.org/wp-content/uploads/2015/02/PC104_Spec_v2_6.pdf (visited on 05/15/2025).

[Pre00] Lutz Prechelt. "An Empirical Comparison of Seven Programming Languages". In: *Computer* 33 (Nov. 2000), pp. 23–29. DOI: 10.1109/2.876288.

[pxn24] GitHub-User "pxntus". *Puslib*. https://github.com/pxntus/puslib. 2024.

[RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005. ISBN: 0321112296.

[Sch+19] Artur Scholz, Jer-Nan Juang, Peter Mader, Jesper Schlegel, and Milenko Starcik. "SpaceCAN - A low-cost, reliable and robust control and monitoring bus for small satellites". In: *Acta Astronautica* 161 (2019), pp. 1–11. ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2019.05.010. URL: https://www.sciencedirect.com/science/article/pii/S0094576519302450.

[SM23] Tomasz Szewczyk and Kostas Marinis. "Standardization concepts for CubeSat applications". In: *2023 European Data Handling & Data Processing Conference (EDHPC)*. 2023, pp. 1–5. DOI: 10.23919/EDHPC59100.2023.10396512.

[Spa] Space Applications Services. *Yamcs Mission Control*. URL: https://yamcs.org/ (visited on 06/15/2025).

[Uta] Utah State University. *GASPACS - Get Away Special Passive Attitude Control Satellite*. URL: https://www.usu.edu/physics/gas/projects/gaspacs (visited on 06/15/2025).

[Zha24] Yuwei Zhang. "Comparative study of the execution efficiency of Python and C++—Based on topological sorting". In: *Applied and Computational Engineering* 34 (2024), pp. 13–17. DOI: 10.54254/2755-2721/34/20230288.